FEATURE SEARCH IN BIOLOGICAL SEQUENCE DATA: ANALYSIS OF GENE-FINDING TOOLS AND IMPLEMENTATION OF INTERACTIVE PATTERN SEARCH

by

JIAN WANG

(Under the Direction of Eileen T. Kraemer)

ABSTRACT

Genome projects continue to produce large quantities of sequence data. Annotation of this sequence data to indicate the location of genes, start and stop codons, inverted and direct repeats, and other patterns of interest is a challenging problem. In this thesis we present three contributions to solving this problem. First, we have performed an analysis of several gene-finding programs for the fungus *N. crassa*, and applied both standard metrics and new metrics we have defined. Next, we have developed a general tool that can automatically evaluate any gene-finding program and report performance metrics. Finally, we have developed an Interactive Pattern Search Tool (IPST) to facilitate finding complex patterns in nucleotide sequence data. The hashtable based approach employed in IPST is compared with the suffix tree approach for pattern search. IPST is applied to the problems of locating Long Terminal Repeat (LTR) retrotransposons and Miniature Inverted repeat Transposable Elements (MITEs) in rice sequences.

INDEX WORDS: Gene-finding analysis, Pattern search, Repeats, LTR retrotransposon, Hashtable, Suffix tree

FEATURE SEARCH IN BIOLOGICAL SEQUENCE DATA: ANALYSIS OF GENE-FINDING TOOLS AND IMPLEMENTATION OF INTERACTIVE PATTERN SEARCH

by

JIAN WANG

B.S., Wuhan University, P.R.China, 1998

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2004

FEATURE SEARCH IN BIOLOGICAL SEQUENCE DATA: ANALYSIS OF GENE-FINDING TOOLS AND IMPLEMENTATION OF INTERACTIVE PATTERN SEARCH

by

JIAN WANG

Major Professor: Eileen T. Kraemer

Committee: Liming Cai

Russell Malmberg

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
August 2004

ACKNOWLEDGMENTS

I would like to thank my major professor Dr. Eileen Kraemer for her constant help, encouragement, and support throughout my graduate study at The University of Georgia. I also appreciate Drs. Russell Malmberg and Liming Cai for serving on my committee, their ideas that initiated the IPST project and their enormous help during my work on this project. I am also grateful to the following people who have used my IPST tool on their experimental data and provided me with great feedback and suggestions: Dr. Ning Jiang, Dr. Mary Bedell and Renyi Liu.

TABLE OF CONTENTS

Page

CHAPTER 1

INTRODUCTION

1.1. Bioinformatics

This thesis involves work in the area of bioinformatics. As stated in [Bayat 2002], bioinformatics is the “application of tools of computation and analysis to the capture and interpretation of biological data”. It is an interdisciplinary area involving biological sciences, computer science, mathematical science, physics, and medicine [Bayat 2002].

As the vast amount of biological data produced by many genome projects continues to grow, the need for computational analysis, organization and management has emerged. Bioinformatics tools have been developed to study the genes and proteins in various organisms, including humans. The importance of bioinformatics in assisting researchers to understand the mechanisms of diseases, design new drugs and develop new treatments is obvious.

Bioinformatics projects center around understanding the structure, function and regulation of genes and proteins. Bioinformatics can basically be divided into two important sub-disciplines [Bioinformatics Factsheet 2001]:

1. In order to efficiently store, access, query and manage large amounts of biological data, tools such as databases have been developed and implemented.

2. Analysis of the vast amount of biological data using algorithmic and statistical methods to unveil and evaluate the relationships among members of the data sets. For instance, gene-finding programs have been developed to locate genes in genomic sequence data, and clustering algorithms have been used to study gene regulation from microarray data.

Recently, with the completion of genome sequencing of quite a few model organisms, such as human, mouse, rat, and two fruit flies, and more sequencing projects being conducted for more organisms, a vast amount of sequence data has become available. The annotation of the sequence data to unveil the gene structures, various types of repeats and other complex biological patterns, such as the Long Terminal Repeat (LTR) retrotransposons and Miniature Inverted repeat Transposable Elements (MITEs), has increasingly emerged as a both important and challenging problem.

Aimed at the above annotation problem, this thesis work provides three contributions:

1. An analysis of several gene-finding programs for the fungus *Neurospora crassa*.
2. The development of a general tool that can automatically evaluate any gene-finding program and will report both the standard metrics and new metrics that we define.
3. The development of a tool, Interactive Pattern Search Tool (IPST), to facilitate finding complex patterns in nucleotide sequence data.

These contributions are introduced in the following sections.

1.2. Analysis of Gene-finding programs for *N. crassa*

Computational gene-finding plays an important role in genome projects. A large number of programs employing various algorithms have been developed to address this problem. The optimal values of parameters for gene-finding programs are often organism-specific. A gene-finding program that performs well for one organism does not necessarily produce good results in another organism. Choosing the right program to find genes in a newly sequenced genome has been a pivotal issue.

Our work on the analysis of gene-finding programs for *N. crassa* is aimed at the above issue using the species *N. crassa* as the case study. We have conducted the evaluation of four commonly used gene-finding programs (GenScan [Burge and Karlin, 1997], GeneMark [Borodovsky and McIninch, 1993], HMMGene [Krogh 1997], and Pombe [Chen and Zhang, 1998]) and a program developed in the Kraemer lab: FFG (Find Fungal Gene). While FFG is designed specifically for the organism *N. crassa*, the other four programs are gene-finding programs designed mainly for other organisms. We have used five manually annotated sequences in the evaluation process.

The performance of the gene-finding program is measured at the exon level using a previously-defined evaluation methodology [Burset and Guigo, 1996]. The performance of those five programs on *N. crassa* is compared.

Selecting the best gene-finding program or programs for a new organism or category of sequences can be time-consuming and error-prone, as well as problematic. In the process of executing these programs on our test sequences, collating the results of the various programs, and calculating statistics, we became keenly aware of the time-consuming and error-prone nature of this process and the variation in reporting methodologies. The need for a standard tool to perform such studies seems clear. We aim to produce an environment and tools to support the task of evaluating gene-finding programs. Toward that goal we have developed a general tool that can automatically evaluate any gene-finding program and will report both the standard and newly defined metrics.

Chapter 2 contains our paper entitled "An analysis of gene-finding programs for *N. crassa*" [Kraemer et al. 2002] and published in the journal *Bioinformatics*. In this paper, we present a general introduction to the problem of computational gene-finding, the underlying algorithms of the gene-finding programs that we evaluated, the test data sets, the evaluation methodology, and the results. Chapter 3 contains our paper entitled "GFPE: gene-finding program evaluation" [Wang et al. 2003] and published in the journal *Bioinformatics* as well. This paper describes the initiative that led to the work of creating a general tool to automatically analyze various gene-finding programs, the usage of the program, and future work related to this tool.

1.3. Interactive Pattern Search Tool (IPST)

Genome projects have produced and continue to produce vast quantities of sequence data. Exploring various patterns contained in these sequences is now the primary concern. Examples of such patterns include direct repeats, inverted repeats, reverse complements, and other more complex structures, such as the long terminal repeat (LTR) retrotransposon elements and miniature inverted repeat transposable elements (MITEs). Given the important roles that these complex patterns may have played in both evolution and regulation of genes and proteins, the need for an efficient computational algorithm to identify and locate these patterns has emerged in the research community. Our tool is designed to specifically address this problem.

This tool is designed to facilitate finding complex patterns in nucleotide sequence data in an interactive manner. In addition to finding direct and inverted repeats, the tool is also capable of performing approximate string search, locating start and stop codons and any combination of the above operations. Users can use a combination of the functions that IPST provides to locate various patterns that they are interested in. We have applied IPST to find LTR (Long Terminal Repeat) retrotransposons and MITEs (Miniature Inverted-repeats Transposable Elements) in rice sequence data.

IPST can be used for multiple sequences and thus is suitable for genomic projects. IPST is implemented in the Java programming language and provides a graphical user interface

and visualization and interaction techniques that focus on interactive exploration of patterns in sequences.

IPST is implemented using a hashtable based approach. A hashtable storing polynucleotide information of the sequences is pre-computed and used in the searching processes. The time taken for building the hashtable allows the fast retrieval of patterns in the sequences during the pattern search processes. As a test for efficiency of the time and memory usage of our hashtable based pattern search approach, we have compared the hashtable based algorithm with a suffix tree based approach, which was also implemented in IPST.

Chapter 4 discusses related work to IPST. Chapter 5 provides a discussion of IPST that represents a manuscript in preparation for submission. Chapter 6 gives a conclusion and states the future work.

CHAPTER 2

AN ANALYSIS OF GENE-FINDING PROGRAMS FOR *NEUROSPORA CRASSA*[1]

[1] Wang, J.[*], Kraemer, E.[*], Guo, J., Hopkins, S., Arnold, J. 2001. *Bioinformatics*. 17(10): 901-912. Reprinted here with permission of publisher.
[*] : Co-authors. Order switched from the original publication with permission of publisher and both authors

Abstract

Motivation: Computational gene identification plays an important role in genome projects. The approaches used in gene identification programs are often tuned to one particular organism, and accuracy for one organism or class of organism does not necessarily translate to accurate predictions for other organisms. In this paper we evaluate five computer programs on their ability to locate coding regions and to predict gene structure in Neurospora crassa. One of these programs (FFG) was designed specifically for gene-finding in Neurospora crassa, but the model parameters have not yet been fully "tuned", and the program should thus be viewed as an initial prototype. The other four programs were neither designed nor tuned for N. crassa.

Results: We describe the data sets on which the experiments were performed, the approaches employed by the five algorithms: GenScan, HMMGene, GeneMark, Pombe and FFG, the methodology of our evaluation, and the results of the experiments. Our results show that, while none of the programs consistently performs well, overall the GenScan program has the best performance on sensitivity and ME(Missing exons) while the HMMGene and FFG programs have good performance in locating the exons roughly. Additional work motivated by this study includes the creation of a tool for the automated evaluation of gene-finding programs, the collection of larger and more reliable data sets for N. crassa, parameterization of the model used in FFG to produce a more accurate gene-finding program for this species, and a more in-depth evaluation of the reasons that existing programs generally fail for N. crassa.

Availability: Data sets, the ffg program source code, and links to the other programs analyzed are available at:

http://jerry.cs.uga.edu/~wang/genefind.html

Contact:eileen@cs.uga.edu

2.1 Introduction

Computational gene identification plays an important role in genome projects. Numerous programs have been developed to address this problem. Some of these programs predict protein-coding regions in genomic DNA sequences, while others predict a set of spliceable exons, or explicitly assemble genes. The methods used in these programs include use of hidden Markov models, linear discriminant analysis, and probabilistic models of gene structure that rely on features such as compositional differences and signals.

In this paper we evaluate several commonly used computer programs designed to predict the structure of protein coding genes in DNA sequences. Some of these algorithms must be "trained" for a particular organism. Thus, the quality of the prediction strategies employed in these programs can vary from organism to organism. Despite these limitations, existing methods of gene prediction and models of gene structure are often applied to newly sequenced organisms, for which no model or method has yet been tuned. Thus, it is important to assess the accuracy of these methods when applying them to a new organism. Here, we wish to evaluate the ability of these programs to accurately

predict gene structure for a particular organism, *Neurospora crassa*, an organism of interest as a well-studied representative of the filamentous fungi. Thus, no previously defined data set is available that meets our needs. However, related experiments have been performed [Fickett and Tung, 1992; Singh and Krawetz 1994; Lopez, *et al*., 1994; Snyder and Stormo, 1995; Burset and Guigo 1996], and we draw upon the methodology applied in these studies.

In weighing and applying the results presented here, the reader must be aware of the methodology involved. A critical element of the type of work we describe is the location of a "good data set". Ideally, this data set would consist of a large, representative set of experimentally-verified annotations. As stated above, such a data set does not yet exist.

Instead, we have relied upon existing sets of annotated sequences, some of which have been annotated using the programs we wish to evaluate. A few sequences exist for which more "manual" means were employed, involving the location of Open Reading Frames (ORFs) and consensus regulatory sequences, BLAST analysis, and matching cosmid sequences with cDNA sequences. Let us refer to the annotations produced by these methods as "actual", and those produced by the programs we evaluate as "predicted". Note that the "actual" annotations do not necessarily correspond to the "true" annotations (experimentally verified). Thus, we run the risk of creating what one reviewer refers to as a "devil's circle". That is, instead of evaluating the results of these programs against the "true" annotations, we instead evaluate them against a set of "actual" annotations that may have been influenced by the programs that we wish to evaluate. Even if the

annotations against which we evaluate the programs have been produced by some other program completely unrelated to the program we wish to evaluate, we still are not able to evaluate the program's ability to predict the "true" set of annotations. Rather, any evaluation we perform will measure a program's ability to produce annotations that correlate well with the method used to produce the "actual" annotations. Unfortunately, this is the nature of the beast in performing such studies. If an adequate set of experimentally verified sequences were to already exist, it is likely that we would be in a stage of study with the organism of interest in which we would no longer need the computational gene prediction tools we seek to create. In summary, the reader should be cautioned that the results presented here represent the correlation of the predictions of these five programs with annotations produced by the methods described. As additional experimentally verified sequences become available, the set of "actual" annotations will change, and our perception of the quality of each of these programs for finding genes in *N. crassa* will change with that. Thus, studies such as the one we describe should be periodically repeated, with the gap between the "true" annotations and what we use as "actual" annotations in our studies gradually closing.

2.2 Systems and Methods

Sequences

In this evaluation we compare the results of five gene prediction programs on five manually annotated sequences. Of these, three annotated sequences were obtained from

the PEDANT web site[PEDANT 2001], one sequence from the University of New Mexico[Bean *et al.*, 2001] and a cosmid sequence H123E02 from the University of Georgia[Kelkar *et al.*, 2001].

The PEDANT database, compiled at the Munich Information Center for Protein Sequences (MIPS)[PEDANT 2001], contains a detailed annotation of *Neurospora* gene models for many of the sequences. The sequenced cosmids and BACs are subjected to an elaborate, manually supervised and evaluated annotation routine. The annotation process [Mannhaupt, 2000] involves BLAST searching (using human and arabidopsis matrices), as well as the application of several separate gene-prediction programs, including GenScan[Burge and Karlin, 1997], GeneFinder[Sulston *et al.*, 1992] and GeneMark[Lukashin and Borodovsky, 1998]. Further evidence from EST matches and from the structure of predicted protein matches is used to create a "corrected" gene, which is reported on the web site. Curators of the site note on their web pages that the training of gene modeling programs for *Neurospora* is still under way. Therefore, for the PEDANT automatic processes they had to use a default setting in the gene prediction for eukaryotes, and they note that these programs may fail to produce a reliable gene prediction using these settings. We looked at three contig sequences for which gene predictions were available: b9j10 (66923 bp, 15 genes), 2a23 (36732 bp, 10 genes), and 4e5 (16820 bp, 3 genes).

Also evaluated was a 36 kilobase-pair cosmid insert, representing genomic DNA from *N. crassa*. This sequence was obtained from Natvig and Nelson in the Department of

Biology at the University of New Mexico, where it was sequenced and characterized [Bean *et al.*, 2000]. The sequenced region contains homologs to *SNZ1* and *SNO1* from *Saccharomyces cerevisiae*, and possesses at least 13 protein-coding genes. The cosmid, G6G8 from the Orbach/Sachs cosmid library, was obtained from the Fungal Genetics Stock Center at the U. of Kansas Medical Center, Kansas City, grown, subcloned, and sequenced according to the procedure described in [Bean *et al.*, 2000]. Basecalling was performed using Phred[Ewing *et al.*, 1998], vector screened using Crossmatch[Green, 1996], and then assembled into contiguous fragments using Phrap[Green, 1996]. The sequence was then annotated and deposited in GenBank (accession number AF309689).

The sequence analysis procedure involved using MacDNASIS v 3.2 to find Open Reading Frames (ORFs) using the codon bias for *N. crassa*, searching for consensus sequences associated with translational start sites and intron splicing, and using BLAST to compare with protein and nucleotide databases at NCBI. Many putative ORFs were eliminated from consideration because they overlapped verified genes; none of the ORFs excluded from the list exhibited a strong pattern of *N. crassa* codon preference. In total, thirteen putative protein-coding genes were predicted in this sequence. Eleven of these putative genes were verified by identification of a homologous sequence using BLAST search. One putative gene was verified by its length (encoding 426 amino acids without interruption) and its strong *N. crassa* codon bias. One more tentative gene was verified by matching it to a *N. crassa* cDNA sequence.

The fifth sequence, H123E02, sequenced in the Arnold Lab at the University of Georgia, is a 54,728bp cosmid sequence that complements the qa-2 mutation of *N. crassa*, and has been previously analyzed and predicted to contain 12 genes [Kelkar *et al.*, 2001]. In our study, we annotated this sequence using a procedure similar to that used in annotating the Natvig sequence. In particular:

1. MacVector TM 7.0 was used to locate all possible ORFs.

2. The Gribskov codon preference plotting method was used with the codon usage table for *N. crassa* to find those ORFs that have a low likelihood of being in a coding region. These ORFs were removed from the original ORF list.

3. Sequence files were created for each of the ORFs in the list. In order to verify the putative ORFs, a small computer program was written to search for those consensus regulatory sequences involved in transcription [Bruchez *et al.*, 1993a] and translation [Bruchez *et al.*, 1993b]. ORFs containing fewer than 7 of the 8 consensus sequences (see Table 2.2) were removed from the list.

4. BLAST searches were performed for ORFs longer that 500 bp, and each gene verified by comparison with BLAST analysis. If two or more ORFs were found to overlap, the ORF representing a verified gene and in the correct frame was kept and the other ORFs were removed from the list.

	Sequence	Minimum match
CAAT box(a)	CAAAT	4
TATA box(a)	TATATAA	5
Plus 1 Sequence consensus(a)	TCATCANC	6
Polyadenylation signal(a)	AATAAA	5
Intron Splicing 5' signal(a)	G_{51}-$G_{99}T_{99}(A_{77}/G_{17})(A_{50}/C_{23})G_{94}(T_{76}/C_{15})$	5
Lariat signal(a)	$(G_{45}/A_{37})C_{94}T_{94}(A_{48}/G_{40})A_{93}C_{82}$	5
Intron splicing 3' signal(a)	"G_4"$(A_{56}/T_{20})(T_{62}/C_{33})A_{100}G_{100}$-$G_{40}$	4
Kozak Sequence(b)	$C_{57}NNNC_{77}A_{81}(A_{44}/C_{43})$"$T_3$"$A_{99}T_{100}G_{99}G_{51}C_{53}$	9

Table 2.1 – Some consensus sequences in *N. crassa* and their acceptance criteria(ζ)

1. ζ: the criteria we used in accepting a given sequence as a consensus sequence; the given sequence must match the minimum number of nucleotides with the consensus sequence to be accepted.

2. a. Bruchez *et al.,* 1993a; b. Bruchez *et al.*,1993b

3 The subscript number indicates the percent occurrence of the particular nucleotide.

4. Symbol "-" indicates the splicing site.

5. If a nucleotide is quoted, it indicates the conserved absence of that particular nucleotide.

5. Because this sequence had been previously annotated by other means, we compared our results with the published results, and found that a gene, qa-2, was missing. We then returned to the complete ORF list originally generated by the MacVector program, and selected those ORFs that reside in the region between the sequences that flank qa-2 in the published map[Kelkar et al., 2001]. Searching with BLAST on these ORFs succeeded in locating the ORF for the qa-2 gene, which had been eliminated in one of the screens in prior steps.

6. To avoid missing a gene, we then added back to the ORF list any predicted ORF of length greater than 1000 bp not in the current list. BLAST searches on these newly added long ORFS did not produce any significant hits. Thus, they were again removed from the ORF list.

7. BLASTX [Gish and States, 1993] was used to locate the exons in the verified ORFs.

The ORFs and exon structure deduced using this method are shown in Table 2.1.

Gene Prediction Programs

Five programs were evaluated, GenScan[Burge and Karlin, 1997], HMMGene[Krogh 1997], GeneMark[Borodovsky and McIninch, 1993], Pombe[Chen and Zhang, 1998], and FFG (Find Fungal Gene), developed at the University of Georgia and described here. Although FFG was designed specifically for gene-finding in *Neurospora crassa,* the

ORF	Protein identification	No. of amino acid	Best blast hit organism	E-Value	frame	Exon locations	Method of exon identification
1	Putative pyridoxal kinase	309	*Schizosaccharomyces pombe*	3e -11	1	3393 - 3566 3579 - 3683 3690 - 3815 3834 - 3905	BLASTX
2	Catabolic 3-Dehydroquinase(qa-2)	173	*Neurospora crassa*	2e -78	1	12550 - 13068	BLASTX
3	Quinate 5-dehydrogenase (qa-3)	321	*Neurospora crassa*	e -179	1	16697 - 17614	BLASTX
4	Quinic Acid Utilization Activator qa-1F	816	*Neurospora crassa*	0	1	25001 - 27448	BLASTX
5	Hypothetical protein SPAC1F12.09	554	*Schizosaccharomyces pombe*	2e -15	1	44189 - 44377 44387 - 44524 44537 - 44617 44627 - 45001	BLASTX
6	Elongation factor 1-Beta(EF-1-BETA)	227	*Xenopus laevis*	6e -33	3	48578 - 48805	BLASTX
7	Fatty acid transport protein	643	*Cochliobolus heterostrophus*	4e -80	-1	40595 - 40413 40391 - 39768 39729 - 39603 39596 - 39429	BLASTX
8	Rehydrin protein homolog	243	*Candida albicans*	5e -46	-3	37332 - 37207 37122 - 36802	BLASTX
9	Regulatory protein ral2	611	*Schizosaccharomyces pombe*	9e -44	-1	32885 - 32820 32810 - 32646 32621 - 32484 32369 - 32241 32216 - 31698 31607 - 31569 31562 - 31449	BLASTX
10	Hypothetical protein YGR277c	305	*Saccharomyces cerevisiae*	2e -12	-1	29187 - 29128 29088 - 29062 29043 - 28933 28806 - 28615 28581 - 28498	BLASTX
11	Quinate Repressor qa-1s	918	*Neurospora crassa*	0	-1	23601 - 21073	BLASTX
12	Quinate Permease (qa-Y)	537	*Neurospora crassa*	0	-1	19916 - 18306	BLASTX
13	3-dehydroshikimate dehydratase (qa-4)	359	*Neurospora crassa*	0	-1	14704 - 13628	BLASTX
14	Hypothetical protein qa-x	340	*Neurospora crassa*	e -121	-1	11329 - 10649	BLASTX

Table 2.2 – ORF and exon structures in *N. crassa* cosmid H123E02

model parameters have not yet been fully "tuned", and the program should thus be viewed as an initial prototype. The other four programs were neither designed nor tuned for *N. crassa*.

GenScan[Burge and Karlin, 1997] is a general-purpose gene identification program that analyzes genomic DNA sequences from a variety of organisms including human, other vertebrates, invertebrates and plants. For each sequence, the program applies a probabilistic model of the gene structure and compositional properties of the genomic DNA for the given organism to determine the most likely gene structure. This model includes consensus sequences involved in transcription and translation, length distributions, and compositional differences. GenScan identifies complete intron/exon structures of a gene in genomic DNA, is able to predict multiple genes, can deal with both partial and complete genes, and can predict consistent sets of genes that occur on either or both strands of DNA. The GenScan program may be accessed through: http://genes.mit.edu/GENSCAN.html. Parameter settings include a choice of organism (vertebrate, arabidopsis, or maize) and a suboptimal exon cutoff value (1.0, 0.50, 0.25, 0.10, 0.05, 0.02, and 0.01). In our evaluation, we used arabidopsis at a cutoff value of 1.0.

HMMGene[Krogh, 1997] is a program for prediction of genes in anonymous DNA, designed for prediction of vertebrate and *C. elegans* genes. The program predicts whole genes, and can be used on whole cosmids or even longer sequences. It can also predict splice sites and start/stop codons. If some features of a sequence are known, such as hits

to ESTs, proteins, or repeat elements, these regions can be locked as coding or non-coding and then the program will find the best gene structure under these constraints. The program is based on a hidden Markov model, a probabilistic model of the gene structure. HMMgene can also report the *n* best gene predictions for a sequence. This is useful if the there are several equally likely gene structures and may even indicate alternative splicing. HMMgene takes an input file with one or more DNA sequences in FASTA format. It also has a few options for changing the default behavior of the program. The output is a prediction of partial or complete genes in the sequences. The output specifies the location of all the predicted genes and their coding regions and scores for whole genes as well as exon scores. The HMMgene program is available at: http://www.cbs.dtu.dk/services/HMMgene. Through the web page, users may enter sequences, select an organism(vertebrate or *C. elegans*), specify whether or not to predict signals, and specify the number of predictions (1 –5) to report. In our evaluation, we specified *C. elegans,* did not predict signals, and reported the best prediction.

The **GeneMark** gene prediction software takes several forms. The original GeneMark program [Borodovsky & McIninch, 1993] relied on inhomogeneous Markov chain models of both coding and non-coding regions, based on analysis of known genes and on the Bayes decision making function, to predict genes in *E. coli* DNA sequences, and was then retrained for *H. influenzae*, *M. genitalium*, and other organisms. GeneMark-Genesis, developed for analysis of organisms such as *M. jannaschii* and *H. pylori*, was designed for the situation in which no experimentally studied segments are available for training. The GeneMark.hmm algorithm [Lukashin and Borodovsky, 1998] generates a

maximum-likelihood parse of the DNA sequence into coding and non-coding regions, and is designed to more precisely locate the exact gene boundaries. This program is available both through e-mail servers (at Georgia Tech and the EMBL Outstation of the European Bioinformatics Institute (EBI)) and through several web pages. Links to the web server, instructions for the e-mail server, and other related information may be found through the GeneMark home page: http://genemark.biology.gatech.edu/GeneMark.

We evaluated both the e-mail server for GeneMark and the GeneMark.hmm program at: http://dixie.biology.gatech.edu/GeneMark/eukhmm.cgi.

Through the e-mail server for GeneMark, the options specified were "spombe" (*S. pombe*) for the organism and exon for the orflist option; otherwise, default values were accepted. Through the web server for GeneMark.hmm, *A. thaliana* was specified for the organism. In analyzing the results of these two programs we found that better results were obtained with GeneMark.hmm using A. thaliana as the model organism. Thus, we report only those results here. Note that we considered using the GeneMark.hmm server at: http://dixie.biology.gatech.edu/GeneMark/whmm.cgi because it has an option for "low eukaryotes" and provides *S. cerevisiae,* which is similar to *N. crassa,* as a model organism. However, we found that the output of this version does not provide exon/intron boundary information.

The **Pombe** program was developed to find genes and predict exon-intron structure in *Schizosaccharomyces pombe* [Chen and Zhang, 1998]. In developing the program, the authors first extracted a training data set from GenBank, checked the annotations for accuracy, and removed redundancy. Execution of the program involves a number of

linear discriminant analyses. For example, one analysis differentiates between {sites, introns, exons} and {pseudo sites, pseudo introns, pseudo exons}. Initiation sites, donor sites, and acceptor sites are identified. Exon and intron predictions are the result of the combination of three linear discriminant functions. Other factors considered include oligonucleotide preferences, positional triplet preferences, and the location of open reading frames. The results of these intermediate analyses are then combined through dynamic programming to predict gene structure. Pombe is freely available for academic use and is available through the web site at**:** **http://argon.cshl.org/genefinder/Pombe/pombe.htm**

Find Fungal Gene (**FFG**) is a pattern-directed program for gene-finding in *Neurospora crassa*, based on statistical analysis of sequence features with genes from *N. crassa* performed by Edelman and Staben[Edelman and Staben, 1994], and conversations with Staben. This study found that sequence features such as translation initiation sites, codon usage in open reading frames, intron length, exon length, intron donor sites, intron branch points, and intron acceptor sites within genes from *N. crassa* are distinctive.

Specifically, coding regions were found to have higher GC content and to exhibit a bias toward codons in which the last nucleotide is C, with a secondary preference for G. Also, the stop codon UAA is more commonly used than either UAG or UGA. An ATG initiator codon and surrounding consensus sequence (CAMMATGGCT) were identified. Most *N. crassa* genes were found to have at least one intron. Introns also tended to be short, with average length 63, median length 70, and a range from 52 to 691 bases. Exon

length varied more widely, from 3 to 5367, with an average length of 509 and median length of 148. Consensus sequences identified for the 5'donor site, splice branch sites, and 3'acceptor sites are G^GTAAGTnnYCnYY, WRCTRACMnnnnnnYY, and WACAG^, respectively [Edelmann and Staben, 1994].

The FFG algorithm begins by identifying possible start and stop sites, as well as left (5' donor) sites, center (splice branch) sites and right (3' acceptor) sites. Frame numbers are associated with start and stop sites. Any subsequence matching the pattern "GTRNGT" is identified as a potential left site; any subsequence matching the pattern "CTRAC" is identified as a potential center site; and any subsequence matching the pattern "YAG" is identified as a potential right site.

Then, the algorithm traverses the list of start sites and builds a list of "primitive" ORFs (Open Reading Frames). Each ORF ends at the first stop site encountered in the same reading frame in the sequence. At this point, each ORF has one exon.

Next, the algorithm repeatedly traverses the ORF list. For each ORF, the algorithm examines the last exon in its list and attempts to extend the ORF to include another exon. This is possible if a splice site can be found within the exon. That is, if the exon contains a "left" (5' donor) site and both a "center" (branch site) and "right" (3' acceptor) site can be found within an acceptable distance (currently set to 300 base pairs). If these are located, another exon is added to the list for that ORF; otherwise, the ORF is marked as complete. Extension terminates when all ORFs are marked as complete.

Finally, the algorithm deletes ORFs that are less than 300bp in length (an ORF less than 300 bases is not likely to be a gene). When several ORFs overlap, the longest one is selected and the others are deleted. The reverse complement strand is then generated and the process repeated.

FFG accepts input sequences in FASTA or plain text format, and produces output in the GFF format [Sanger Center: GFF, 2000], a sequence annotation format developed with gene finding in mind. A more highly tuned FFG program that uses a genetic algorithm to tune parameters such as the required homology to the consensus sequences, the relative weights of each of the sites, lengths, and distances, and including dinucleotide composition and codon bias, is under development, and will make use of the evaluation performed in this study.

The Evaluation Methodology

In our study, we evaluated only the accuracy of the prediction, and did not evaluate factors such as execution time or memory requirements. In general, prediction accuracy can be measured at three levels: at the level of the coding nucleotide, at the level of exonic structure, and the level of the predicted protein product. At the protein product level, the protein encoded by the actual gene is compared with the protein encoded by the predicted gene. We focus our evaluation on the exon level, but have developed and apply here a technique that provides combined information about both ability to predict

sequence coding regions and how well signals are identified, which we explain later in this section.

Measurements of accuracy at the coding level compare predicted coding value with the actual coding value for each nucleotide along the test sequence. In this widely used approach predictions are divided into four categories:

- True Positive = nucleotides classified as coding in both actual and predicted (TP).
- True Negative = nucleotides classified as non-coding in both actual and predicted (TN).
- False Positive = classified as coding in predicted, but as non-coding in actual (FP).
- False Negative = classified as non-coding in predicted, but as coding in actual (FN).

Sensitivity is defined as the proportion of coding nucleotides that have been correctly predicted as coding. That is, sensitivity = TP / (TP + FN). Specificity is the proportion of noncoding nucleotides that have been correctly predicted as non-coding. That is, specificity = TN/(TN+FP). An issue that arises in evaluating specificity is that the frequency of noncoding nucleotides in genomic DNA sequences is much greater than the frequency of coding nucleotides, so that TN tends to be much larger than FP, with the result of a tendency toward very large non-informative values for specificity. Thus, in much of the literature on gene structure prediction, specificity is instead defined to be TP/(TP + FP), the proportion of predicted coding nucleotides that are actually coding.

Other commonly used metrics based on these categories are the Correlation Coefficient(CC), defined as:

$$CC = \frac{(TP \times TN) - (FN \times FP)}{\sqrt{(TP + FN) \times (TN + FP) \times (TP + FP) \times (TN + FN)}}$$

the Simple Matching Coefficient(SMC), defined as:

$$SMC = \frac{TP + TN}{TP + FN + FP + TN}$$

and the Average Conditional Probabilty(ACP), defined as:

$$ACP = \frac{1}{4}\left[\frac{TP}{TP + FN} + \frac{TP}{TP + FP} + \frac{TN}{TN + FP} + \frac{TN}{TN + FN}\right]$$ [Burset and Guigo, 1996].

While nucleotide-level metrics are often used to evaluate how well the program locates sequence coding regions, exonic structure metrics are typically used to evaluate how well the sequence

signals (splice sites, start codons, and stop codons) are identified [Burset and Guigo, 1996]. Our evaluation focused on measuring the accuracy of predictions at the exon level, by comparing predicted and actual exons along the test sequence. Although this approach is widely used, no unique criterion has been used to consider an exon as "correctly" predicted. The strictest criterion would score an exon prediction as a correct

match only if an exact match exists between actual and predicted start and stop locations, both splicing boundaries correctly identified. We label these as “type 1” predictions. A looser criterion scores a prediction as correct if a partial match occurs, if at least one of the splice sites has been correctly identified. We label these as “type 2” predictions. Finally, a predicted exon may be scored as correct if the overlap between actual and predicted exceeds some threshold. We label these as “type 3” predicted exons. While the first two approaches are more stringent, the advantage of the third approach is that an evaluation performed using this method provides combined information about both the ability of the program to locate sequence coding regions and how well sequence signals are identified. Type 4 predicted exons do not overlap with any actual exon.

The notions of sensitivity and specificity are still applicable in measurements performed at the exon level. Sensitivity is the proportion of actual exons in the test sequence that are correctly predicted. Specificity is the proportion of predicted exons that are correctly predicted. Also useful are the notions of Missing Exons(ME) and Wrong Exons(WE). Missing Exons indicates the proportion of actual exons with no overlap to predicted exons. Wrong Exons indicates the proportion of predicted exons with no overlap to actual exons.

Determining the criteria to use in selecting a threshold for the type 3 exons proved challenging. To address this problem, we developed a method of selecting a threshold for overlap between actual and predicted exons that relies on the notions of Overlap-sensitivity and Overlap-specificity and an initial empirical evaluation.

From all of the predicted exons obtained by running all of the programs on four of the five test sequences (annotations for H123E02 were not yet available) we selected all the type 2 and preliminary type 3(any overlap at all) exons. For each of these exons, we calculated the overlap-sensitivity, overlap-specificity, and combined overlap percentage. Overlap-sensitivity is the number of nucleotides in the overlapping region between the predicted exon and the actual exons, divided by the number of nucleotides in the actual exon. Overlap-specificity is the number of nucleotides in the overlapping region, divided by the number of nucleotides in the predicted exon. A Combined Overlap Percentage (COP) was defined to be (OverlapSn + OverlapSp)/2. We then divided the exons into different groups based on the value of COP, such as group 100, [95,100), [90,95), ... , [0,10). Then we calculated the fractions of the exons falling into each of the above groups and drew a curve (figure 2.1) with the y-axis representing the COP value and the x-axis representing the fraction of exons with a COP value equal to or greater than the corresponding y value. As can be seen from figure 2.1, the “knee” of the curve falls between 70 and 90 on the x-axis, and appears to be linear in this range. Based on this curve, “greater than 80%" was determined to be a reasonable threshold to define a type 3 exon. In reporting the results of our evaluations, we define three categories, labeled one-star (*), two-star (**), and three-star (***). The one-star category includes only the type 1 exons. The two-star category includes only the type 1 and type 2 exons. Both of these categories may be used to evaluate the ability of a program to exactly locate exon and intron boundaries. The three-star category combines this information with a measure of the ability of a program to correctly predict coding regions, and consists of type 1 exons,

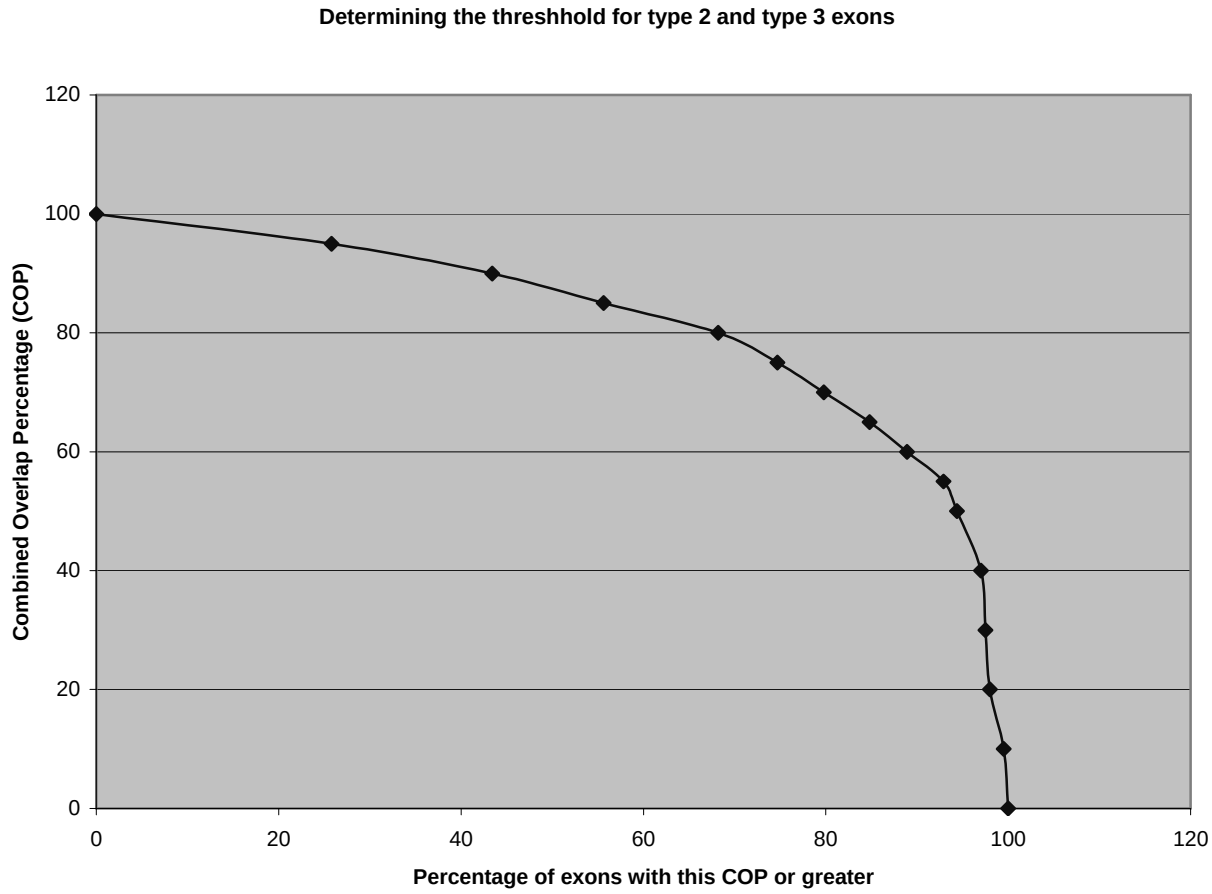


Figure 2.1 – Chart of data used to select “appropriate” value of COP for type-2 and type-3 threshold cutoff.

type 2 exons for which the COP exceeds the threshold (80%), and type 3 exons for which the COP exceeds the same threshold.

2.3 Results and discussion:

Annotation of the H123E02 cosmid sequence:

Our annotations for the *N. crassa* cosmid H123E02 sequence are shown in Table 2.1. As described in the Systems and Methods section, we used a method similar to that employed in the annotation of the Natvig sequence. All of the 12 genes predicted by [Kelkar *et al.*, 2001] are "recovered" in our study, and two additional hypothetical genes (Hypothetical protein SPAC1F12.09 and Hypothetical protein YGR277c) are predicted. We note, however, that in the absence of Kelkar's annotations, only 11 of those 12 genes would have been recovered. Specifically, the Catabolic 3-Dehydroquinase(qa-2) gene would have been missed, indicating that the methodology described may have some room for improvement. Two elements of the screening process are possible culprits in the omission of the ORF that represents the Catabolic 3-Dehydroquinase. One possibility is that it was eliminated because it exhibited a low codon preference when using the Gribskov codon preference plotting method. Although the Gribskov method can help to locate highly and moderately expressed genes very well, and can save substantial effort in analyzing unlikely ORFs, it is not good at distinguishing weakly expressed genes from non-genes [Gribskov *et al.*, 1984]. Thus, those ORFs that were removed because of low Gribskov plotting likelihood might represent weakly expressed genes.

The other possibility is that the qa-2 gene was removed during the process of verifying the presence of the consensus sequences. In this study, eight consensus sequences, seen in Table 2.2 along with the acceptance criteria: the CAAT box, the TATA box, the +1 sequence consensus, the polyadenylation signal sequences, the intron splicing 5' signal, the lariat sequence, the intron splicing 3' signal[Bruchez and *et al.*, 1993a] and the Kozak sequence[Bruchez and *et al.*, 1993b] were searched for in those putative ORFs. The screen for these consensus sequences is fairly stringent, and thus some actual ORFs may have been eliminated.

To locate exons within the ORFs, we simply used the BLASTX [Gish and States, 1993] method to deduce the exon boundaries. Ideally, we would have incorporated the information derived from aligning the cDNA clone with the ORFs. In that way, the boundaries would be more precisely located.

Prediction accuracy analysis for Natvig Sequence

Results of analysis for the Natvig test sequence are shown in table 2.3. For this sequence, it seems that regardless of the method used to define the exon, the GenScan program has the best sensitivity followed by the FFG and GeneMark.hmm programs. For specificity, the FFG program behaves best, for all definitions of an actual exon.

Prediction accuracy analysis for the H123E02 cosmid

The result for the cosmid H123E02 is shown in table 2.4. None of the five programs predicted exons that have the same boundaries as the actual exons. Since an important deviation of our hand annotation procedure from that used in the annotation of the Natvig sequence is that we did not compare and align those putative ORFs with the cDNA sequence, it illustrates the importance of cDNA sequence in the hand annotation procedure. As a consequence, the three-star(***) category is the most informative for this sequence. The results in table 2.4 show that, for this sequence, none of the programs performed well and all programs performed similarly in sensitivity except that HMMGene's score for sensitivity is relatively low. The FFG program has a relatively better specificity.

Prediction accuracy analysis for the contig b9j10

For this test sequence, the result shown in table 2.5 indicates that the GenScan program has the best sensitivity and specificity. It also has relatively low ME and WE. The GeneMark.hmm program also has a relatively good sensitivity and the HMMGene program has a relatively good specificity. We note that the GenScan program was used in the annotation process for the PEDANT sequences (b9j10, 2a23, 4e5), thus biasing the results with these sequences in favor of the GenScan program.

	GenScan	HMMGene	GeneMark.hmm	Pombe	FFG
# of correct exons*	14	3	7	1	8
# of correct exons**	24	7	15	8	17
# of correct exons***	24	8	19	7	17
# of actual exons	27	27	27	27	27
# of predicted exons	82	27	93	57	40
# of missing exons	1	15	5	9	8
# of wrong exons (Type 4)	56	16	70	42	19
Sn*	0.52	0.11	0.26	0.04	0.3
Sn**	0.89	0.26	0.56	0.3	0.63
Sn***	0.89	0.3	0.7	0.26	0.63
Sp*	0.17	0.11	0.08	0.02	0.2
Sp**	0.29	0.26	0.16	0.14	0.43
Sp***	0.29	0.3	0.2	0.12	0.43
ME	0.04	0.56	0.19	0.33	0.3
WE	0.68	0.59	0.75	0.74	0.48

Table 2.3 - Prediction accuracy analysis for Natvig Sequence

* : Only include Type 1 exons

** : Include both Type 1 and Type 2 exons

***: Include Type 1 and only those Type 2 and

Type 3 with Combined Overlap Percentage

(COP) greater than 80%

	GenScan	HMMGene	GeneMark.hmm	Pombe	FFG
# of correct exons*	0	0	0	0	0
# of correct exons**	1	2	2	2	5
# of correct exons***	9	5	8	7	7
# of actual exons	34	34	34	34	34
# of predicted exons	112	44	110	75	49
# of missing exons	3	6	6	2	14
# of wrong exons (Type 4)	95	29	96	61	39
Sn*	0	0	0	0	0
Sn**	0.03	0.06	0.06	0.06	0.15
Sn***	0.26	0.15	0.24	0.21	0.21
Sp*	0	0	0	0	0
Sp**	0.01	0.05	0.02	0.03	0.1
Sp***	0.08	0.11	0.07	0.09	0.14
ME	0.09	0.18	0.18	0.06	0.41
WE	0.85	0.66	0.87	0.81	0.8

Table 2.4 - - Prediction accuracy analysis for the H123E02 cosmid

* : Only include Type 1 exons

** : Include both Type 1 and Type 2 exons

***: Include Type 1 and only those Type 2 and
Type 3 with Combined Overlap Percentage
(COP) greater than 80%

Prediction accuracy analysis for the contig 2a23

The result for the contig 2a23 is shown in Table 2.6, and indicates that the GenScan program has the best sensitivity and the lowest ME. The FFG program has a relatively good performance on specificity for this sequence.

Prediction accuracy analysis for the contig 4e5

The result for this test sequence is shown in Table 2.7. The GenScan program shows the best performance on sensitivity and ME. In regard to specificity, although the HMMGene program is not good at locating the exons exactly, it is good at roughly locating the exons with the highest Sp** and Sp***.

2.4 Summary

The average results for evaluating the prediction accuracy on these five test sequences is shown in table 2.8. Overall, the GenScan program has the best performance on sensitivity and ME. But as for specificity, the HMMGene and the FFG program have good performance in locating the exons roughly, since they both have relatively high average scores on Sp** and Sp***. This result encourages heavier weighting of the factors considered by GenScan and HMMGene in the parameterized method being developed for the refined FFG program.

	GenScan	HMMGene	GeneMark.hmm	Pombe	FFG
# of correct exons*	43	6	9	0	4
# of correct exons**	58	13	28	11	19
# of correct exons***	51	16	36	12	18
# of actual exons	61	61	61	61	61
# of predicted exons	123	41	174	102	66
# of missing exons	4	42	18	10	32
# of wrong exons	62	22	131	64	35
Sn*	0.7	0.1	0.15	0	0.07
Sn**	0.95	0.21	0.46	0.18	0.31
Sn***	0.84	0.26	0.59	0.2	0.3
Sp*	0.35	0.15	0.05	0	0.06
Sp**	0.47	0.32	0.16	0.11	0.29
Sp***	0.41	0.39	0.21	0.12	0.27
ME	0.07	0.69	0.3	0.16	0.52
WE	0.5	0.54	0.75	0.63	0.53

Table 2.5 -- Prediction accuracy analysis for the contig b9j10

* : Only include Type 1 exons

** : Include both Type 1 and Type 2 exons

***: Include Type 1 and only those Type 2 and
Type 3 with Combined Overlap Percentage
(COP) greater than 80%

	GenScan	HMMGene	GeneMark.hmm	Pombe	FFG
# of correct exons*	15	0	3	2	4
# of correct exons**	27	9	15	6	12
# of correct exons***	25	12	18	10	14
# of actual exons	28	28	28	28	28
# of predicted exons	73	23	79	49	29
# of missing exons	1	11	5	5	10
# of wrong exons	44	7	56	31	9
Sn*	0.54	0	0.11	0.07	0.14
Sn**	0.96	0.32	0.54	0.21	0.43
Sn***	0.89	0.43	0.64	0.36	0.5
Sp*	0.21	0	0.04	0.04	0.14
Sp**	0.37	0.39	0.19	0.12	0.41
Sp***	0.34	0.52	0.23	0.2	0.48
ME	0.04	0.39	0.18	0.18	0.36
WE	0.6	0.3	0.71	0.63	0.31

Table 2.6 -- Prediction accuracy analysis for the contig 2a23

* : Only include Type 1 exons

** : Include both Type 1 and Type 2 exons

***: Include Type 1 and only those Type 2 and
Type 3 with Combined Overlap Percentage
(COP) greater than 80%

	GenScan	HMMGene	GeneMark.hmm	Pombe	FFG
# of correct exons*	5	1	2	0	1
# of correct exons**	6	4	5	2	3
# of correct exons***	6	3	4	1	3
# of actual exons	7	7	7	7	7
# of predicted exons	24	7	48	24	10
# of missing exons	1	2	2	3	2
# of wrong exons	18	2	43	21	5
Sn*	0.71	0.14	0.29	0	0.14
Sn**	0.86	0.57	0.71	0.29	0.43
Sn***	0.86	0.43	0.57	0.14	0.43
Sp*	0.21	0.14	0.04	0	0.1
Sp**	0.25	0.57	0.1	0.08	0.3
Sp***	0.25	0.43	0.08	0.04	0.3
ME	0.14	0.28	0.28	0.43	0.29
WE	0.75	0.28	0.9	0.88	0.5

Table 2.7 -- Prediction accuracy analysis for the contig 4e5

* : Only include Type 1 exons

** : Include both Type 1 and Type 2 exons

***: Include Type 1 and only those Type 2 and Type 3 with Combined Overlap Percentage (COP) greater than 80%

	GenScan	HMMGene	GeneMark.hmm	Pombe	FFG
Sn*	0.49	0.07	0.16	0.02	0.13
Sn**	0.74	0.28	0.47	0.21	0.39
Sn***	0.75	0.31	0.55	0.23	0.41
Sp*	0.19	0.08	0.04	0.01	0.1
Sp**	0.28	0.32	0.13	0.1	0.31
Sp***	0.27	0.35	0.16	0.57	0.32
ME	0.08	0.42	0.23	0.23	0.38
WE	0.68	0.47	0.8	0.74	0.52

Table 2.8 – Average results of prediction accuracy on all five test sequences

* : Only include Type 1 exons

** : Include both Type 1 and Type 2 exons

***: Include Type 1 and only those Type 2 and Type 3 with Combined Overlap Percentage (COP) greater than 80%

In another study, GeneMark.hmm was suggested as the most accurate exon prediction program for the *Arabidopsis* genome[Pavy, *et al.*, 1999]. In our study, however, GeneMark did not perform as well. This suggests the importance of evaluating programs based on the particular organism that one wishes to study.

The results also show that most of the programs performed better at sensitivity than at specificity. On one hand, this may indicate that our original annotation for these test sequences is too stringent. Alternatively, the gene finding programs may be too liberal in retaining unlikely ORFs and exons.

Although the three test sequences contig b9j10, contig 2a23, and contig 4e5 were annotated using GenScan, thus biasing the results with these sequences in favor of GenScan, we note that the relatively good performance of GenScan is consistent in all of these five sequences. Ideally, those sequences that have been annotated using a testing program should not be used to evaluate that program. However, the dearth of sequences annotated by other means was a major factor in our decision to include these sequences.

In summary, none of the gene-finding programs evaluated consistently performs well at finding genes in *N. crassa.* The programs may have failed because the models they use are inappropriate for this organism, or the models may be appropriate but the model parameters may be inappropriate. Further investigation is necessary to determine the reasons that these programs failed.

The FFG algorithm is designed for *N. crassa.* Model parameters include homology to consensus sequences, relative weights of donor, branch, and acceptor sites, distances between these sites, and lengths of introns and exons. We are working to parameterize the model used in FFG to obtain a more accurate gene-finding program for this species. A program that uses a genetic algorithm has been designed and implemented, with the goal of determining appropriate weights for these parameters in the prediction formula, and for including dinucleotide composition and codon bias. However, this tuning process requires the existence of a "seed" set of reliably annotated sequences. The studies described in this paper were performed in the process of obtaining such a data set, and work on this refined version of FFG continues. Thus, the FFG algorithm evaluated in this paper should be viewed as a prototype version.

We note that we have used only 5 test sequences in our evaluation, and note further that despite the small number of sequences, the evaluation process was quite tedious and time-consuming. To address this problem we are developing a tool to automate the process of performing these studies, permitting rapid evaluation of a set of programs and/or parameters for those programs against a set of annotated sequences. The existence of such a tool will permit the scientist working with a "new" organism to evaluate the ability of existing programs to "correctly" predict (that is, in a way that correlates with the current best annotation procedure) genes in a newly studied organism, and to select the best parameter sets for those programs. Further, we seek to obtain sequences that have been experimentally verified, and to expand our evaluation to include additional

programs (GeneWise and NEX have been suggested). Finally, we plan to perform periodic re-evaluations to include additional programs, as existing programs are updated, and as more and/or more reliable data becomes available.

References

Bean L.E., Dvorachek W.H., Braun, E.L., Errett,A., Saenz, G.S., Giles,M.D., Werner-Washburne, M., Nelson, M.A., and Natvig D.O. .(2001) Analysis of the pdx-1 (snz-1/sno-1) Region of the *Neurospora crassa* Genome: Correlation of pyridoxine-requiring phenotypes with mutations in two structural genes, *Genetics* 157(3):1067-75.

Benian, G., Tinley, T.L., Tang, X., and Borodovsky, M. (1996) The C. elegans Gene unc-89, required for Muscle M-line Assembly, Encodes a Giant Modular Protein Composed of Ig and Signal Transduction Domains, *J. Cell Biology*, 6, 835-848.

Blattner, F.R., Burland, V., Plunkett, G., Sofia, H.J., Daniels, D.L. (1993) Analysis of the Escherichia coli genome. IV. DNA sequence of the region from 89.2 to 92.8 minutes *Nucleic Acids Research*. 21(23): 5408-5417.

Borodovsky, M., Sprizhitsky Y., Golovanov, E. and Alexandrov, A. (1986) Statistical Patterns in Primary Structures of Functional Regions in the E. Coli Genome: I. Oligonucleotide Frequencies Analysis. *Molecular Biology*, 20, 826-833.

Borodovsky, M., Sprizhitsky Y, Golovanov, E. and Alexandrov, A. (1986) Statistical Patterns in Primary Structures of Functional Regions in the E. Coli Genome: II. Non-homogeneous Markov Models. *Molecular Biology*, 20, 833-840.

Borodovsky, M., Sprizhitsky Y., Golovanov, E. and Alexandrov, A. (1986) Statistical Patterns in Primary Structures of Functional Regions in the E. Coli Genome: III. Computer Recognition of Coding Regions. *Molecular Biology*, 20, 1145-1150.

Borodovsky, M. and McIninch J. (1993) GeneMark: Parallel Gene Recognition for both DNA Strands. *Computers & Chemistry*, 17, 123-133.

Borodovsky, M., Koonin, Eu. , and Rudd, K. (1994) New Genes in Old Sequences: A Strategy for Finding Genes in a Bacterial Genome. *Trends in Biochemical Sciences*, 19, 309-313.

Borodovsky, M., Rudd, K. and Koonin, E. (1994) Intrinsic and Extrinsic Approaches for Detecting Genes in a Bacterial Genome. *Nucleic Acids Research*, 22, 4756-4767.

Borodovsky, M., McIninch, J., Koonin, E., Rudd, K., Medigue, C., and Danchin, A.. (1995) Detection of New Genes in the Bacterial Genome Using Markov Models for Three Gene Classes. *Nucleic Acids Research*, 23: 3554-3562.

Bruchez, J.J.P., Eberle, J. and Russo, V.E.A. (1993a) Regulatory sequences in the transcription of *Neurospora crassa* genes: CAAT box, TATA box, introns, poly(A) tail formation sequences. *Fungal Genet. Newsl.* 40: 89 – 96.

Bruchez, J.J.P., Eberle, J. and Russo, V.E.A. (1993b) Regulatory sequences involved in the translation of *Neurospora crassa* mRNA: Kozak sequences and stop codons. Fungal Genet. Newsl. 40: 85-88.

Bult, C.J., White, O., Olsen, G.J., Zhou, L., Fleischmann, R.D., Sutton, G.G., Blake, J.A., FitzGerald, L.M., Clayton, R.A., Gocayne, J.D., Kerlavage, A.R., Dougherty, B.A, Tomb, J.F., Adams, M.D., Reich, C.I., Overbeek, R., Kirkness, E.F., Weinstock, K.G, Merrick, J.M., *et al.*, (1996). Complete genome sequence of the methanogenic archaeon, Methanococcus jannaschii. *Science*, 273(5278), 1058-1073.

Burge C. and Karlin S. (1997) Prediction of complete gene structures in human genomic DNA. *J Mol Biol*. 268(1):78-94.

Burland V., Plunkett G., Daniels D.L., Blattner F.R. (1993) DNA Sequence and Analysis of 136 Kilobases of the Escherichia coli Genome: Organizational Symmetry Around the Origin of Replication. *Genomics* 16, 551-561.

Gish, W. and States, D.J. (1993) Identification of protein coding regions by database similarity search. *Nat Genet.* 3(3):266-72.

Burset M and Guigo R. (1996) Evaluation of gene structure prediction programs. *Genomics* 34(3):353-67

Chen, T. and Zhang M. Q., (1998) Pombe: a gene-finding and exon-intron structure prediction system for fission yeast. Yeast 14: 701-710.

Ewing, B., Hillier, L. Wendl, M. and Green, P. (1998) Base-calling of automated sequencer traces using Phred. *Genome Research,* 8, 175--185

Fickett J.W. and Tung C.S.(1992) Assessment of protein coding measures, *Nucleic Acids Res.* 20:6441-50.

Fleischmann, R. D., Adams, M. D., White, O., Clayton, R.A., Kirkness, E.F., Kerlavage, A.R., Bult, C.J., Tomb, J.F., Dougherty, B.A., Merrick, J.M. and *et al.,* (1995) Whole-genome random sequencing and assembly of Haemophilus influenzae Rd. *Science* 269:496-512.

Fraser, C. M., Gocayne, J. D., White, O., Adams, M.D., Clayton, R.A., Fleischmann, R.D., Bult, C.J., Kerlavage, A.R., Sutton, G., Kelley, J.M., and *et al.,* (1995) The minimal gene complement of Mycoplasma genitalium. *Science* 270:397-403.

Green, P. (1996) Documentation for Phrap. *Nature* 390:580-586.

Gribskov, M., Devereux, J., and Burgess, R.R. (1984) The codon preference plot: graphic analysis of protein coding sequences and prediction of gene expression. *Nucleic Acids Research*. 12: 539-549.

Hayes, W., and Borodovsky, M. (1998) How to interpret an anonymous bacterial genome: machine learning approach to gene identification. *Genome Res.*, 8(11), 1154-1171.

Hayes, W. S., and Borodovsky, M. (1998) Deriving Ribosome Binding Site (RBS) statistical models from unannotated DNA sequences and the use of the RBS model for N-terminal prediction in Proceedings of Pacific Symposium on Biocomputing 1998. *World Scientific*, 279-290.

Hirosawa M., Kaneko T., Tabata, S., McIninch, J.D., Hayes, W., Borodovsky, M., Isono, K. (1995) Prediction of the Coding Regions in a Contiguous 1MB Sequence of the Genome of Synechocystis sp. Strain PCC6803. *DNA Research*, 2, 239-246.

Kelkar, H. S., Griffith, J., Case, M.E., Covert, S.F., Hall, R.D., Keith, C.H., Oliver, J.S., Orbach, M.J., Sachs, M.S., Wagner, J.R., Weise, M.J., Wunderlich, J.K., Arnold, J.

(2001). The *Neurospora crassa* genome. Cosmid Libraries Sorted by Chromosome. *Genetics* 157(3):979-90.

Krogh. A. (1997) Two methods for improving performance of an HMM and their application for gene finding, *In Proc. of Fifth Int. Conf. on Intelligent Systems for Molecular Biology, ed.* Gaasterland, T. et al., Menlo Park, CA: AAAI Press, 1997, pp. 179-186.

Lopez, R., Larsen, F., and Prydz, H. (1994), Evaluation of the exon predictions of the GRAIL software", *Genomics*, 24, 133-136.

Lukashin, A.V. and Borodovsky, M. (1998) GeneMark.hmm: new solutions for gene finding. *Nucleic Acids Research*, 26, 1107-1115.

Mannhaupt, G.,(2000) Sequence Analysis of the *Neurospora crassa* Genome. *IBC's Third International Symposium on Fungal Genomics*. July 27-28, 2000. (Oral presentation)

McIninch, J., Hayes W., and Borodovsky, M. (1996) Applications of GeneMark in Multispecies Environments , *ISMB '96 Proceedings*, 176-188.

PEDANT (2000) Genome analysis and annotation at MIPS, http://pedant.mips.biochem.mpg.de/#MIPS

Sanger Center: GFF(2000) GFF(General Feature Format) Specifications Document, http://www.sanger.ac.uk/Software/formats/GFF/GFF_Spec.shtml.

Singh, G. B., and Krawetz, S.A. (1994), Computer based exon detection: an evaluation metric for comparison, *International Journal of Genome Research*, 1, 321-338.

Snyder E.E, Stormo G.D.(1995) Identification of protein coding regions in Genomic DNA. *J Mol Biol* 248(1):1-18

Sulston, J., Du, Z., Thomas, K., Wilson, R., Hillier, L., Staden, R., Halloran, N., Green, P., Thierry-Mieg, J., Qiu, L., *et al.*, (1992), The C. elegans genome sequencng project: A beginning. *Nature*, 356:37-41

Suyama, M., Nagase, T. and Ohara, O. (1999) HUGE: a database for human large proteins identified by Kazusa cDNA sequencing project. *Nucleic Acids Research*, 27, 338-339.

Tatusov R.L., Mushegian A.R., Bork, P., Brown, N.P., Hayes, W.S., Borodovsky, M., Rudd, K.E., Koonin, E.V. (1996) Metabolism and Evolution of H. influenzae Deduced from Whole Genome Comparison to E. coli, *Current Biology*, 6, 279-291.

CHAPTER 3

GFPE: Gene-finding Program Evaluation[2]

[2] Wang, J. and Kraemer, E. 2003. *Bioinformatics*. 19(13): 1712-1713.
Reprinted here with permission of publisher.

Abstract

Summary: GFPE (Gene-finding program evaluation) is a set of Java classes for evaluating gene-finding programs. A command-line interface is also provided. Inputs to the program include the sequence data (in FASTA format), annotations of "actual" sequence features, and annotations of "predicted" sequence features. Annotation files are in the GFF (General Feature Format) promoted by the Sanger center. GFPE calculates a number of metrics of accuracy of predictions at three levels: the coding level, the exon level, and the protein level.

Availability: The program is free, available at: ftp://anonymous@iubio.bio.indiana.edu/molbio/genefind/

Contact: eileen@cs.uga.edu

3.1 The program

Computational gene identification plays an important role in genome projects, and numerous programs have been developed to address this problem. Selecting the best gene-finding program or programs for a new organism or category of sequences can be time-consuming and error-prone, as well as problematic for the following reasons: 1) The approaches used in gene identification programs are often tuned to one particular organism; accuracy for one organism or class of organism does not necessarily translate to accurate predictions for other organisms. 2) The performance of the gene-finding programs may depend on the parameter settings used to perform the analysis. 3)

Published evaluations of gene identification programs are often not only limited to a particular organism, but may report only a subset of the available metrics. This use of different metrics by the authors of different gene-finding programs complicates the comparison of results. The effort required to reproduce these studies to verify the results or to generate a consistent set of metrics is typically prohibitive.

Despite these limitations, existing methods of gene prediction and models of gene structure are often applied to newly sequenced organisms, for which no model or method has yet been tuned. Thus, it is important to have a rapid and reliable means to assess the accuracy of different gene identification methods and parameter settings when beginning a new genome project or evaluating a new gene identification program.

Recently, we evaluated several commonly used gene-prediction programs to compare the ability of these programs to accurately predict gene structure for a particular organism, *Neurospora crassa* (Kraemer, 2001). In the process of executing these programs on our test sequences, collating the results of the various programs, and calculating statistics, we became keenly aware of both the time-consuming and error-prone nature of this process and the variation in reporting methodologies in prior studies.

The need for a standard tool to perform such studies seems clear. We aim to produce an environment and tools to support the task of evaluating gene-finding programs. Toward that goal we have developed a set of Java classes to perform the necessary analysis and a

simple command-line program through which they may be invoked. A graphical user interface and analysis environment is under development.

The evaluation criteria are based on those described in (Burset et al. 1996). New criteria for prediction at the exon level have been added (Kraemer, 2001). The evaluation is carried out at three levels: the coding level, the exon level and the protein level.

The GFPE program takes as input the DNA sequence file in FASTA format, the GFF-formatted sequence annotation and gene-finding program output. GFF is the short for Gene-Finding Format or General Feature Format. The attribute fields are <seqname> <source> <feature> <start> <end> <score> <strand> <frame> [attributes] [comments] (Sanger Center, GFF, 2000). For those genefinding programs (GenScan, GeneMark, and Pombe) whose output is not in GFF format, some Java programs are included in the GFPE package to convert their outputs to GFF format. In addition, since the exon positions are used in the accuracy calculations in all three levels, the <feature> attribute field must contain the string “exon” for recognition. GFF-format output that contains a feature name other than “exon” can be converted using a Java program included in the GFPE package.

Figure 3.1 shows an example output of the GFPE program on a single sequence. Notations used in this example are largely as illustrated in (Burset et al. 1996). New notation describes prediction accuracy at the exon level. If both splicing boundaries are correctly identified, this prediction is defined as “type 1”. If at least one of the splicing

boundaries from the prediction matches with the correct exon, this prediction is defined as “Type 2”. “Type 3” predictions are those predicted exons whose overlap with the actual exons exceeds some threshold. In figure 3.1, for prediction at the exon level, those marked with “*” mean that only “Type 1” predictions are included in the calculation. For those marked with “**”, both “Type 1” and “Type 2” predictions are included. Those marked with “***” include “Type 1” predictions and those “Type 2” and “Type 3” predictions whose sensitivity and specificity values exceed some threshold (Kraemer 2001). Note that GFPE can be used to evaluate prediction accuracy not only for a single sequence but also for multiple sequences. Users provide a file in which each line contains the name of a sequence file, and the annotation file and gene-finding program output file for that sequence. The average prediction accuracy across multiple sequences is calculated on a weighted basis.

Execution of the analysis codes of a 17kb sequence for 4 gene-finding programs required 90 seconds on a Pentium II 450 Mhz PC with 128 MB of RAM running Red Hat Linux 7.2. On the same machine, execution of the analysis codes on 10 sequences whose average size was 17kb required approximately 160 seconds.

This GFPE program saves the evaluators of gene-finding programs substantial effort in calculating the prediction accuracy. Note that GFPE can be used to evaluate the prediction accuracy of gene-finding programs on a whole genome basis. Limitations of the program include the difficulty of executing the various gene-finding programs to be evaluated. To solve this problem, we are developing a GUI and environment to simplify

```
[wang@elaine GFPE]$ java Run dat/4e5-seq dat/4e5_anno.gff

dat/4e5_FFG_out.gff
Sequence name = dat/4e5-seq
Annotation file name = dat/4e5_anno.gff
Perdition File Name = dat/4e5_FFG_out.gff
The length of this testing sequence is: 16820
This genefinding program has the following prediction accuracy with this testing sequence in the coding level:

TP : 3619  TN : 26587   FP : 1796   FN : 1638   Sn : 0.68841547   Sp : 0.6683287
Correlation Coefficient (CC) : 0.617672  Simple Matching Coefficient (SMC) : 0.8979191
Average Conditional Probability (ACP) : 0.8088583     Approximate Correlation (AC) : 0.6177166

***************

This genefinding program has the following prediction accuracy in the exon level:

# of correct exons* is: 1    # of correct exons** is: 3
# of correct exons*** is: 3   # of actual exons is: 7
# of predicted exons is: 10   # of missing exons is: 2
  # of wrong exons(Type4) is: 5
Sn* : 0.14285715    Sn** : 0.42857143     Sn*** : 0.42857143     Sp* : 0.1     Sp** : 0.3     Sp*** : 0.3
 (Sn* + Sp*)/2 : 0.12142857   (Sn**+Sp**)/2 : 0.3642857
(Sn*** + Sp***)/2 : 0.3642857     ME : 0.2857143   WE : 0.5

***************

The prediction accuracy at the protein level is: 0.060139433
```

Figure 3.1 Example output of the GFPE program on a single sequence

and/or automate the process of executing the runs of the gene-finding programs in the sequences of interest and of collecting and analyzing the results.

References

Burset,M. and Guigo,R.(1996) Evaluation of gene structure prediction programs. *Genomics,* 34, 353-367.

Eileen Kraemer, Jian Wang, Jinhua Guo, Samuel Hopkins, Jonathan Arnold. **An analysis of gene-finding programs for *Neurospora crassa*.** *Bioinformatics,* 17(10): 901-912, October 2001

Sanger Center: GFF (2000) GFF (General Feature Format) Specifications Document. http://www.sanger.ac.uk/Software/formats/GFF/GFF_Spec.shtml

Sun Microsystems: The Source for Java[TM] Technology. http://java.sun.com/

CHAPTER 4

RELATED WORK

The Interactive Pattern Search Tool (IPST) that we have developed is designed for pattern searching in biological sequences. The functions that are provided in this tool include approximate string searching, finding direct and inverted repeats and combinations of these operations. Before we describe the IPST tool in detail, we first discuss related work in string searching, which is the central element in this tool. Below we will talk about five categories of string matching and the related problems of exact pattern match, approximate pattern match, finding direct repeats, finding inverted repeats and finding the reverse-complement type inverted repeats.

4.1. Exact pattern match

(In the analysis of the time and space complexity of each algorithm, unless otherwise specified, m and n are the lengths of the pattern and the text, respectively).

In Gusfield's book [Gusfield, 1997], the exact matching problem is given the following definition:

"Given a string *P* called the pattern and a longer string *T* called the text, the **exact matching** problem is to find all occurrences, if any, of pattern *P* in text *T*."

As an example, the pattern “agct” is found at positions 1, 7 and 12 in the text “agctatagctcagct”. Several occurrences of the pattern may overlap in the text.

The exact pattern match is a very important issue in many areas. In the biological sciences, exact pattern match is needed to conduct biological database search for some specific DNA or protein sequences. In library sciences, exact pattern match is used for searching the catalog for the books or articles that the users want.

As the amount of information in texts increases, the time that it takes to search for specific patterns in the text could become burdensome. The efficiency of exact pattern matches has been a very important issue.

A number of methods have been developed to address the exact string matching problem.

(i) The naïve method

The simplest way to deal with the exact string match problem begins by aligning the left end of the pattern string with the left end of the text string. Then the characters of the pattern and the text are compared from left to right. If all characters match, the pattern is found in the text. Otherwise, the pattern is moved to the right by one more position. The comparison is made repeatedly until the right end of the pattern reaches the end of the text string.

The naïve method is very straightforward and easy to implement. However, the worst-case running time is $\Theta(nm)$ (where n and m are the lengths of the pattern and the text, respectively), which is unsatisfactory.

(ii) The modified naïve method with preprocessing

The naïve method is simple but not efficient. A modified approach based on the naïve method emphasizes preprocessing of the pattern, which is basically to compute the $Z_i(s)$ values for each $1 < i < |s|$ (s is the string).

$Z_i(S)$ is defined to be the length of the longest substring of X that starts at i and matches a prefix of S, given a string S and a position i > 1.[Gusfield, 1997].

This method utilizes the pre-computed $Z_i(s)$ values and constructs a new String S = P$T, where $ is a symbol that appears neither in P nor in T. Let n = |P| and m = |T|. In order to solve the exact matching problem, we need only to find if for any value $i > n+1$, a $Z_i(s) = n$ can be found.

As proved in [Gusfield, 1997], computing the $Z_i(S)$ can be solved in $O(m)$ time. Thus, this method can solve the exact matching problem in $O(m)$ time. As discussed in [Gusfield, 1997], this approach can be implemented to use only $O(n)$ space.

(iii) The Boyer-Moore algorithm

This method employs three clever ideas that were missed in the naïve method: right-to-left scan, the bad character rule and the (strong) good suffix rule. In the preprocessing stage, three variables: $L'(i)$, $l'(i)$, and R(x) (i is the position index of the pattern and x is each character in the alphabet) are pre-computed. For each i, $L'(i)$ is defined as the largest position less than n so that string P[$i..n$] matches a suffix of P[1.. $L'(i)$] and the character preceding the suffix is not P(i-1). If no position satisfies both of these conditions, $L'(i)$ will be zero. $l'(i)$ is the length of the largest suffix of P[$i..n$] and this suffix is also a prefix of the pattern P. If there is not such a suffix, $l'(i)$ is defined to be zero. R(x) is defined to be the position of the right-most occurrence of character x (a character in the alphabet) in the pattern P. If x is not in P, R(x) will be zero [Gusfield, 1997]. The algorithms to compute $L'(i)$, $l'(i)$, and R(x) are available in [Gusfield, 1997].

Here is an outline of the algorithm from [Gusfield, 1997].

```
{Preprocessing stage}
        Given the pattern P,
        Compute L'(i) and l'(i) for each position i of P,
        and compute R(x) for each character x Є Σ.
{Searching stage}
        k := n;
        while k <= m do
                begin
```

```
        i:=n;
        h:=k;
        while i > 0 and P(i) = T(h) do
                begin
                i:= i-1
                h:=h-1;
                end;
I       f i = 0 then
                begin
                report an occurrence of P in T ending at position k.
                k := k+n-l'(2);
                end
        else
                shift P (increase k) by the maximum amount determined by
                the (extended) bad character rule and the good suffix rule.
        end;
```

In the preprocessing phase, both the time and space complexity are $O(m+\Sigma)$, where Σ is the alphabet size. The searching phase is in $O(mn)$ time complexity [Charras et al. 1997].

(iv) The Knuth-Morris-Pratt algorithm

This best known linear-time algorithm for the exact string matching problem was developed by Knuth, Morris and Pratt [Knuth et al., 1977].

At first, several terms need to be defined.

$sp_i(P)$ is defined to be the longest proper substring of P[1..i] that ends at i and that matches a prefix of P. Consequently, $sp_i'(P)$ is defined in the same way but with the added condition that characters P(i+1) and $P(sp_i'+1)$ are unequal. Then we have the definition of the failure function F'(i), which is defined to be $sp'_{i-1} + 1$ for each position i from 1 to n+1, where sp_0' and sp_0 are defined to be zero (also define $F(i) = sp_{i-1}+1$) [Knuth et al., 1977].

Here is an outline of the Knuth-Morris-Pratt algorithm from [Gusfield, 1997].

```
begin
preprocessing P to find F'(k) = sp_{k-1}'+1 for k from 1 to n+1.
    c :=1;
    p :=1;
    while c + (n-p) <= m
    do begin
        while P(p) = T( c ) and p <= n
        do begin
            p:= p+1;
            c:= c+1;
```

end;

if $p = n + 1$ then

report an occurrence of P starting at position $c - n$ of T.

if $p := 1$ then $c := c+1$

P:=$F'(p)$;

end;

end.

It has been shown that finding find $F'(k)$ can be solved in linear time [Gusfield, 1997]. Overall, in the preprocessing phase, both the time and space complexity are O(m). The searching phase is in $O(n+m)$ time complexity [Charras et al. 1997].

This algorithm can be easily implemented. A C-program implementation is available from the web [Charras et al. 1997].

(v) The Shift-Or algorithm

In [Charras et al. 1997], the procedure of this method is described as the following:

Let R be a bit array of size m. Vector R_j is the value of the array R after text character y[j] has been processed. It contains information about all matches of prefixes of x that end at position j in the text for 0 < i <= m-1. If x[0,i] = y[j-i, j], then $R_j[i] = 0$; otherwise, it is 1.

For each $R_j[i] = 0$, $R_{j+1}[i+1] = 0$ if x[i+1] = y[j+1]; else, it is 1. $R_{j+1}[0] = 0$ if x[0] = y[j+1]; else, it is 1.

The computation of R_{j+1} can be reduced to two bit operations: a shift and an or: R_{j+1}=SHIFT(R_j) OR $S_y[j+1]$, where S_c contains the positions of the character c in the pattern. $S_c[i]$=0 if and only if x[i]=c, where i is the index of each character in the pattern.

This algorithm is efficient. The preprocessing phase is in $\boldsymbol{O}(m+\sigma)$ time (where $\mathcal{T}$ is the alphabet size) and space complexity and the searching phase is in $\boldsymbol{O}(n)$ time complexity(m is the size of the pattern and n is the size of the text). The only catch to this method is that the pattern length should be shorter than the memory-word size of the machine.

A C-program implementation is available from the web [Charras et al. 1997].

(vi) Karp-Rabin algorithm [Charras et al. 1997]

This method uses a hashing function to compare the hashing values of the patterns and each substring in the text. If the values differ, then no match exists. Otherwise, a character-to-character comparison is needed to confirm the match.

This method uses the following hashing function:

hash(w[0 .. m-1])=(w[0]*2^{m-1}+ w[1]*2^{m-2}+⋯+ w[m-1]*2^{0}) mod q,

where m is the length of the word w and q is a large number.

and

rehash(a,b,h)= ((h-a*2^{m-1})*2+b) mod q

This hashing function is computationally efficient, highly discriminating for strings and hash(y[j+1 .. j+m]) can be easily computed from hash(y[j .. j+m-1]) and y[j+m]: hash(y[j+1 .. j+m])= rehash(y[j], y[j+m], hash(y[j .. j+m-1]).

The preprocessing phase is in ***O***(m) time and space. The searching phase has a worst-case time ***O***(mn) and a ***O***(n+m) expected time.

A C-program implementation is available from the web [Charras et al. 1997].

4.2. Approximate pattern match

(In the analysis of the time and space complexity of each algorithm, unless otherwise specified, m and n are the lengths of the pattern and the text, respectively; k is the number of mismatches allowed).

Approximate pattern (string) match allows errors when matching the pattern with the text. The number of errors allowed is specified by the user.

In computational biology, searching a specific subsequence over a long DNA or protein sequence is fundamental to primer design, sequence alignment, homology study, etc. Since the usual length of the target sequence (or text) is vast and the patterns rarely match the text exactly, the application of approximate string match is more widely appreciable than exact string match.

In addition to computational biology, approximate string match is also a pivotal issue in many other areas such as signal processing, text retrieval, handwriting recognition, and image compression [Navarro, 2001].

(i). The dynamic programming approach

The edit distance of two strings is the smallest number of steps to convert one string to another. In the dynamic programming approach, the edit distance should be calculated and filled into a matrix at first. The matrix $C_{0..|x|,0..|y|}$ is computed as follows:

$C_{i,0} = i$; $C_{0,j} = j$;

$C_{i,j} = C_{i-1,j-1}$ if $x_i = y_j$

$1 + \min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1})$, otherwise.

The matrix of edit distance between the strings “survey” and “surgery” is shown in figure 4.1.

To use this method for approximate string match, we must allow any text start position to be a potential start of a match. We can achieve this by simply setting $C_{0,j} = 0$ for all j €0..n. Then the matrix’s column can be initialized with C_i being set to i and the text is processed character by character. For each new text character T_j, its column vector is updated to $C'_{0..m}$ by the formula:

$C'_i = C_{i-1}$ if $P_i = T_j$

$1+\min(C'_{i-1}, C_i, C_{i-1})$ otherwise.

The text positions can be found where $C_m <= k$ is reported.

		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1	**0**	1	2	3	4	5	6
u	2	1	**0**	1	2	3	4	5
r	3	2	1	**0**	1	2	3	4
v	4	3	2	1	**1**	2	3	4
e	5	4	3	2	2	**1**	**2**	3
y	6	5	4	3	3	2	2	**2**

Figure 4.1 The matrix of edit distance between the strings “survey” and “surgery” [Navarro, 2001].

The search time of this method is ***O***(mn) and its space requirement is ***O***(m) [Navarro, 2001].

(ii) Algorithms aimed at improving the average case

(ii-a) The cut-off heuristic algorithm

Originally developed by [Ukkonen, 1985], this algorithm aims to improve the average case. It was proven to have a ***O***(kn) average running time and ***O***(m) space [Chang and Lampe, 1992; Baeza-Yates and Navarro 1999].

This method takes advantage of the fact that usually a pattern does not match the text and the values at each column of the matrix in the dynamic programming approach quickly

reaches k+1 from top to bottom. If a cell in the matrix has a value greater than k+1, then the result of the search does not depend on its exact value. If a cell has a value of at most k, it will be called “active”. This method keeps counting the last active cell and does not work on the remainder of the cells [Navarro, 2001].

An implementation is available from [Baeza-Yates and Navarro 1999; Navarro, 2001]

(ii-b) Column partitioning algorithm.

Produced by [Chang and Lampe, 1992], this algorithm is based on the property of the dynamic programming matrix that the numbers along each column are normally increasing. “Runs” of consecutive increasing cells are focused on. A run ends when C_{i+1} != C_i+1. A value called loc(j, x) = $\min_{j' >= j} P_{j'}$ = x for all pattern positions j and all characters x must be pre-computed. For each column of the matrix they are trying to find where the run is going to end and thus find the next character match. The run can be performed on all of the columns in parallel.

This algorithm needs an average searching time of ***O***(kn/sqrt(σ)) and a ***O***(mσ) space, where σ is the alphabet size[Navarro, 2001].

(iii) Wu, Manber and Myers' algorithm based on automata [Wu, et al. 1996]

A non-deterministic automaton (NFA) can be used to model approximate search. This automaton was first proposed in [Ukkonen, 1985]. As illustrated in figure 4.2, every row of states denotes the number of errors seen and every column of states stands for matching a pattern prefix. Horizontal arrows from state to state represent matching a character and vertical arrows insert a character in the pattern. Note that they advance in the text but not the pattern. Solid diagonal arrows substitute a character and dashed diagonal arrows delete a character of the pattern. The automaton signals the end of a match when a rightmost state is active.

In the work of [Wu, et al. 1996], they trade time with space by utilizing a Four Russians technique [Arlazarov et al. 1975]. The columns were partitioned into blocks of r cells which took 2r bits each. The transitions from a region to the next region in the column were precomputed.

This algorithm has an average O(kn/logn) time and O(mn/logn) worst time with an O(n) space[Navarro, 2001].

(iv). Bit-parallelism algorithms

These algorithms focus on parallelizing the computation on a bit-wise fashion.

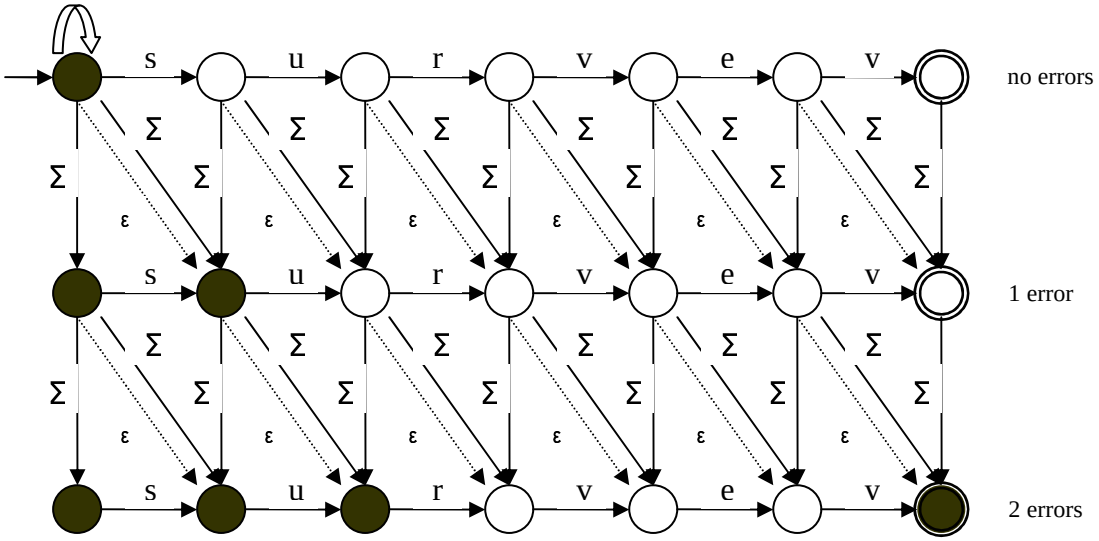


Figure 4.2 The NFA for approximate string matching of the pattern “survey” with two errors. The black states are those active states after reading the text “surgery”. This figure is reproduced from Fig. 15 in [Navarro 2001] with slight modifications.

(iv-a). Wu and Manber's method [Wu et al., 1992].

The idea is to use bit-parallelism to simulate the NFA that we discussed before. Each row i of the NFA fits in a computer word R_i with each state being represented by a bit. Since all of the k+1 computer words (where k is the number of mismatches allowed and these k+1 words each have 0, 1, 2, … k mismatches) have the same structure, parallelism was utilized to simulate all the transitions of the NFA for each new text character using bit operations. The update to obtain the new R'_i values at text position j from the current R_i values is computed in the following way:

$R'_0 = ((R_0 << 1) | 0^{m-1}1) \& B[T_j]$

$R'_{i+1} = ((R_{i+1} << 1) \& B[T_j]) | R_i | (R_i << 1) | (R'_i << 1)$, where k is the number of mismatches allowed and w is the length of each computer word (in bits).

This method takes O(k*ceil(m/w)*n) time both in the worst case and the average case [Navarro, 2001].

(iv-b). Baeza-Yates and Navarro's method [Baeza-Yates et al. 1999]

This method parallelizes the computation even more than the Wu and Manber's method [Wu et al., 1992]. Instead of parallelizing the computation of the rows in the NFA, packing the states of the automaton along the diagonals is adopted. The number of

complete diagonals is m-k+1. D_i is the row of the first active state in diagonal i. The new D'_i values after reading the text position j are computed as follows:

$D'_i = \min(D_{i+1}, D_{i+1}+1, g(D_{i-1}, T_j))$,

$G(D_i, T_j) = \min(\{k+1\} U \{r/r>=D_i \wedge P_{i+r} = T_j\})$

This algorithm has an improved $\boldsymbol{O}(\text{ceil}(k(m-k)/w)*n)$ worst case time and $\boldsymbol{O}(\text{ceil}(k^2/w)*n)$ on average [Navarro, 2001]

(iv-c) Parallelizing the dynamic programming matrix

Instead of parallelizing the rows or diagonals of the automata, this method [Myers, 1999] parallelizes the computation of the dynamic programming matrix.

The differences along columns instead of the columns themselves are represented so that two bits per cell were enough. A new set of recurrences is defined for the horizontal and vertical differences as the following:

$\Delta v_{i,j} = C_{i,j} - C_{i-1,j} = \min(-Eq_{i,j}, \Delta v_{i,j-1}, \Delta h_{i-1,j}) + (1 - \Delta h_{i-1,j})$

$\Delta h_{i,j} = \min(-Eq_{i,j}, \Delta v_{i,j-1}, \Delta h_{i-1,j}) + (1 - \Delta v_{i,j-1})$,

Where $Eq_{i,j}$ is 1 if $P_i = T_j$ and 0 otherwise.

The resulting algorithm has an improved running time. The worse case time is O (⌐m/w¬ n) and the average case time is O (⌐k/w¬ n) [Navarro, 2001].

(v). Filtering algorithms

This is a quite new and still active area in which many algorithms [Tarhio and Ukkonen, 1993; Jokinen et al.1996; Wu and Manber, 1992; Baeza-Yates and Navarro, 1998, 1999; Navarro and Raffinot, 2000; Takaoka, 1994.; Chang and Marr, 1994; Sutinen and Tarhio, 1995] have been developed based on the idea of filtering.

Since the text usually contains much unmatched content, it will be very efficient for the matching process if those unmatched parts in the text are first filtered. The filtering algorithm focuses mainly on improving the average case.

Among all of the above-mentioned algorithms, Chang and Marr's algorithm [Chang and Marr, 1994] has achieved the optimal average case bound: $\boldsymbol{O}$ (n(k+$\log_\sigma$m)/m). The space complexity of this method is $\boldsymbol{O}$ (m^t) for some constant t which depends on σ, which is the alphabet size.

Chang and Marr's algorithm first splits the text in contiguous substrings of length l = t$\log_\sigma$m. It then searches the text substrings of length l in the pattern allowing errors. The best matches allowing errors inside P are pre-computed for every l-tuple. The searches

start at the beginning of the block and continue along the consecutive l-tuples in the pattern until the total number of errors made exceeds k.

(vi) Suffix tree approach

According to [Gusfield, 1997], the suffix tree is formally defined as follows:

"A suffix tree T for an m-character string S is a rooted directed tree with exactly m leaves numbered 1 to m. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of S. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of S that starts at position i. That is, it spells out S [i..m]."

To use the suffix tree approach to solve the exact string match problem, a suffix tree for the text must first be built.

Esko Ukkonen [Ukkonen, 1995] has developed a linear-time algorithm to construct the suffix tree.

Here is a high level description of Ukkonen's algorithm, according to [Gusfield, 1997].

```
construct tree T1.
for i from 1 to m-1 do
begin {phase i+1}
        for j from 1 to i+1
        begin {extension j}
        find the end of the path from the root labeled S[j..i] in the
        current tree. If needed, extend that path by adding character S(i+1),
        thus assuring that string S[j..i+1] is in the tree.
        end;
end;
```

There are three rules to perform a suffix extension. Let S[j..i] = β be a suffix of S[1..i]. In extension j, when the algorithm finds the end of β in the current tree, it extends β to be sure the suffix βS(i+1) is in the tree based on the following three rules [Gusfield, 1997]:

Rule 1: If in the current tree, path β ends at a leaf, then simply add the character S(i+1) to the end of the label on that leaf edge.

Rule 2: If there is no path from the end of string β that starts with S(i+1), but there is at least one labeled path that continues from the end of β, a new leaf edge starting from the end of β must be created and labeled with S(i+1). Plus, a new node will need to be created there if β ends inside an edge.

Rule 3: If some path from the end of string β starts with character S(i+1), do nothing.

This sketch and several additional speedup tricks described in [Gusfield, 1997] account for an algorithm that runs in $\boldsymbol{O}(m)$ time and $\boldsymbol{\Theta}(m\log|\Sigma|)$ space [Gusfield, 1997], where Σ is the alphabet.

Having built the suffix tree, we are now able to fully utilize the advantage of the suffix tree to solve our approximate string matching problem, or the so-called k-mismatch problem.

In [Gusfield, 1997], the following approach is used to solve this problem.

begin

1. Set *j* to 1 and *i'* to *i*, and *count* to 0.

2. compute the length *l* of the longest common extension starting at positions *j* of *P* and *i'* of *T*.

3. if *j*+*l* = *n*+1, then a *k*-mismatch of *P* occurs in *T* starting at *i* ; stop.

4. if *count* <= *k*, then increment *count* by one, set *j* to *j*+l+1, set *i*' to *i*'+*l* +1, and go to step 1.

if *count* = *k*+1, then a *k*-mismatch of *P* does not occur starting at *i*; stop.

end.

Note that in the above algorithm there is an intermediate step for computing the longest common extension, which is a bridge to the k-mismatch problem. The definition and solution of the longest common extension problem are available in [Gusfield, 1997].

The time and space requirements for the *k*-mismatch problem are $\boldsymbol{O}(km)$ and $\boldsymbol{O}(n+m)$, respectively [Gusfield, 1997].

A Java-program implementation for constructing the suffix tree and solving the k-mismatch problem is available from [Dorohonceanu et al. 2000].

(vii). *STAR* – An algorithm to search for approximate repeats [Delgrangey et al. 2004]

STAR is an algorithm to find all significant approximate tandem repeats in a DNA sequence given a motif and the DNA sequence. Here the motif is the pattern and the DNA sequence is the text.

First of all, *STAR* aligns the sequence (say s) with an exact tandem repeat (ETR) of the motif m and obtains an optimal list of mutations that convert the repeat into s and the optimal length for the ETR. The *Wraparound Dynamic Programming (*WDP*)* algorithm [Fischetti et al.1993] is employed in this step.

Secondly, *STAR* involves a compression procedure that outputs a compressed version of the sequence s: s'. The compression is aimed at reducing the size of the sequence by

exploiting a property, which is “s contains segments that are significant approximate tandem repeats (ATRs) of the motif m”. Based on the mutation list created from step 1, *STAR* evaluates the compression gain as if s were a single ATR of m. Theoretically, a true ATR segment has a positive compression gain.

Finally, *STAR* optimizes the global compression gain over s by decompressing s into ATR and non-ATR segments optimally with respect to the global compression gain.

If the lengths of the motif and the sequence are p and n, respectively, the time complexity of this algorithm is $\boldsymbol{O}(np + n\log n)$.

An implementation of this algorithm is available from http://atgc.lirmm.fr/star.

4.3. Finding direct repeats

(In the analysis of the time and space complexity of each algorithm, unless otherwise specified, m and n are the lengths of the pattern and the text, respectively; k is the number of mismatches allowed).

In biology, repeats or so-called tandem repeats in DNA or protein sequences are often of enormous interests to researchers. These repeats consist of two or more adjacent or isolated approximate copies of a pattern of nucleotide or amino acid sequences. It has

been noted that a large part of many genomes consists of repetitive sequences. Repetitive sequences have three major categories:

1. Local repeats (tandem repeats and simple sequence repeats. Note that tandem repeats refer to those contiguous approximate copies of a nucleotide sequence and the number of copies is more that two [Benson 1999]).
2. Families of dispersed repeats (mostly transposable elements and retro-transposed elements).
3. Duplicated genomic fragments [Bao et al., 2002].

These repeats usually allow errors. So, more precisely, they should be called approximate repeats.

Repetitive sequences play an important role in evolution and they are often fundamental regulatory elements in the genome. Some tandem repeats have been shown to cause human diseases. A more specific case of this type of repeats is the Long Terminal Repeat (LTR) in some retrotransposons, which contains two repetitive copies of nucleotide sequence at both ends of the same DNA strand. Since the vastly large number of repetitive sequences in the genome are impossible to be manually analyzed, the need for efficient computational algorithms to find those repeats has emerged in the research community.

(i) Tandem repeats finder [Benson 1999].

This program is designed to find tandem repeats in DNA sequences. The algorithm looks for matching nucleotide sequences separated by a common distance d and looks for k-tuple matches (Note that a k-tuple is a window of k consecutive characters from the nucleotide sequence. Tandem repeats are found through scanning of sequence with a small window, determining the distance between exact matches and testing the statistical criteria.

An implementation of the program with a web interface is available at http://c3.biomath.mssm.edu/trf.html .

(ii) *STRING* – a heuristic approach to find tandem repeats in DNA sequences [Parisi, 2003].

STRING uses a heuristic method to find all possible tandem repeats in DNA sequences. Two heuristic criteria are adopted in this algorithm. First of all, instead of studying the whole sequence, those that are considered to be more promising ones according to an autoalignment procedure are examined as potential tandem repeats. Note that autoalignment is a procedure to search local alignments of a sequence with itself [De Fonzo et al., 1998]. Secondly, instead of studying all possible consensus words, only those that are considered to be more promising (in a way inspired by the autoalignment procedure) are selected.

The algorithm consists of two phases. In the first phase, prospective Tandem Repeats (TRs) are obtained by using the search for autoalignments. In the second phase, the final TR search is performed based on the above heuristic criteria.

An implementation of this algorithm is available from http://www.caspur.it/~castri/STRING/ .

(iii) ***mreps***: efficient and flexible detection of tandem repeats in DNA [Kolpakov et al. 2003]

This program is able to identify all tandem repeats in the whole genome with no limitation on the size of the repeated pattern. It can output tandem repeats with all possible pattern sizes.

The algorithm first utilizes an efficient combinatorial algorithm [Kolpakov et al. 1999, 2001] to find all repetitive structures of a certain kind in a given sequence. This will create a set of raw repeat sequences which will undergo further processing by a series of heuristic treatments: trimming the left and right edges of each repeat, computing the best period and merging for each repeat, filtering out statistically expected repeats and merging repeats with the same period p overlapping by at least 2p.

The time complexity during the combinatorial part of this algorithm is ***O***(*nk*log(*k*)+*S*) for *k*-mismatch tandem repeats (*S* is the number of repeats found) [Kolpakov et al. 2003].

The ***mreps*** program has a web interface accessible through http://www.loria.fr/mreps/ .

(iv). REPEATMASKER [Smit et al. 2003]

This is a program developed to screen DNA sequences for interspersed repeats and low complexity DNA sequences. The program utilizes the program cross_match to perform sequence comparisons. The cross_match program is an efficient implementation of the Smith-Waterman-Gotoh algorithm with some enhancements.

The time complexity of this method is ***O***(nklog(k)log(n)+S) in the case of edit distance and ***O***(nklog(n/k)+S) in the case of Hamming distance, where k is the maximal distance between two tandem repeats, and S is the number of repeats found [Smit et al. 2003].

Implementation of this program is available at [Smit et al. 2003].

(v). EQUICKTANDEM

EQUICKTANDEM is a program from the EMBOSS package [Rice et al. 2000]. It is aimed at finding tandem repeats up to a specified size in DNA sequences. The algorithm

is based on a statistical method. The program is available at http://www.hgmp.mrc.ac.uk/Software/EMBOSS/Apps/equicktandem.html .

(vi). A method to find two specific kinds of tandem repeats in DNA [Hauth et al. 2002].

Regular tandem repeats consist of perfect and degenerate Tandem Repeats, variable length tandem repeats (VLTRs) and multi-period tandem repeats (MPTRs). This method focuses on finding the latter two kinds of regular tandem repeats in DNA sequences.

A variable length tandem repeat (VLTR) is a simple nested tandem repeat in which the copy number for some pattern is variable rather than constant, while a multi-period tandem repeat (MPTR) contains the nested concatenation of two or more *i*-similar patterns, [Hauth et al. 2002], which are patterns that share *i* same characters.

This method to find VLTRs and MPTRs is performed in three major tasks. (1). Determine a tandem repeat's period and its approximate location in order to isolate a tandem repeat. (2). Find the pattern affiliated with a region period. (3). Use the pattern to characterize the region.

Algorithms of this method are available through http://www.cs.wisc.edu/areas/theory/ .

(vii) TROLL – Tandem Repeat Occurrence Locator [Castelo, 2002].

TROLL is designed to find Simple Sequence Repeats (SSRs), which are repetitive short nucleotide sequences with length less than six and that are fairly well conserved.

TROLL requires the user to provide a motif list and finds all occurrences of patterns from the motif list. TROLL is based on the *Aho Corasick* Algorithm (ACA) [Aho and Corasick, 1975], which was aimed at finding all occurrences from a list of patterns in a text. The ACA keeps track of a failure link while encountering a partial match in the text. It uses the failure link to continue the search to avoid re-examining characters in the text. The TROLL implements the ACA to find pre-selected patterns in the text. It also records the tandem repeats such that the SSRs can be located.

The time complexity for TROLL is ***O*** (n+m+k), where n, m and k are the total length of all patterns being searched, the length of the DNA sequence, and the number of occurrences found, respectively.

This program is available through http://finder.sourceforge.net .

4.4. Finding inverted repeats (on the same DNA strand)

This type of repeat refers to those pairs of DNA sequences found in identical but inverted form. Note that the pair of repeats appear on the same strand. Inverted repeats of this kind have appeared in some DNA transposons in plants [Feschotte, 2002].

No literature was found to design algorithms for locating this type of repeat.

4.5. Finding the reverse-complementary type inverted repeats (repeat pairs are on opposite DNA strands)

The pair of repeats also appear in identical but inverted form. In addition, the two copies of the repetitive elements appear on opposite DNA strands. This situation is also called reverse complement. DNA or RNA sequences that contain this type of repetitive elements can form inside loops. This is particularly important in the formation of RNA secondary structures.

No literature was found to design algorithms for locating this type of repeats.

CHAPTER 5

IPST – INTERACTIVE PATTERN SEARCH TOOL[3]

[3] Wang, J. and Kraemer, E. To be submitted to *Bioinformatics*.

Abstract

Motivation: Genome projects have produced and continue to produce vast quantities of sequence data. Exploring various patterns contained in these sequences is now a primary concern. Examples of such patterns include direct repeats, inverted repeats, reverse complements, and other more complex structures, such as the long terminal repeat (LTR) retrotransposon elements and miniature inverted repeat transposable elements (MITEs). Given the important roles that these complex patterns may have played in both evolution and regulation of genes and proteins, the need for an efficient computational algorithm to identify and locate these patterns has emerged in the research community. Our tool is designed to specifically address this problem.

Results: We have designed and implemented a tool called Interactive Pattern Search Tool (IPST) to facilitate finding repeats and other complex patterns in biological sequence data. IPST utilizes a hashtable of n-mers for storage and fast retrieval of various patterns in the sequence. The time-consuming hashtable-building process is compensated for by the fast retrieval of patterns in an interactive manner. In addition to locating direct and inverted repeats, IPST can also be applied to pattern search, locating start and stop codons and a combination of any of these operations. IPST can be used for multiple sequences. IPST is implemented in the Java programming language and provides a graphical user interface and visualization and interaction techniques that focus on interactive exploration of patterns in sequences. In this article, we demonstrate the abilities of IPST to find miniature inverted repeat transposable elements (MITEs) and long terminal repeat (LTR)

retrotransposon elements in rice sequences. In addition, we compare the time and memory efficiency of our hashtable based algorithm in pattern search against an implementation of IPST using a suffix tree approach.

Availability: Program source code, data sets and result are available at http://www.cs.uga.edu/~eileen/IPST

Contact: eileen@cs.uga.edu

5.1 Introduction

Genome projects have produced and continue to produce vast quantities of sequence data. Exploring the information contained in these sequences is now a primary concern. Examples of such patterns include various repetitive sequences, such as direct repeats, inverted repeats and reverse complements, and other more complex structures, such as the long terminal repeat (LTR) retrotransposon elements and miniature inverted repeat transposable elements (MITEs).

Repetitive sequences are those nucleotide sequences appearing in identical form or approximately identical form with similarity above a certain threshold.

Repetitive sequences can be divided into three major categories:

1. Local repeats (tandem repeats and simple sequence repeats. Note that tandem repeats refer to those contiguous approximate copies of a nucleotide sequence in which the number of copies is more that two [Benson 1999]).
2. Families of dispersed repeats (mostly transposable elements and retro-transposed elements).
3. Duplicated genomic fragments [Bao et al., 2002].

In terms of their orientation, repeats can be grouped into two types: direct repeats, which are sequences found in identical form with the same orientation; and inverted repeats, which are sequences in identical form but inverted orientation. For the inverted repeats, the copies of the repeats appear on the same strand of the nucleotide sequence. If one copy of an inverted repeat appears on one DNA strand and the other appears on the complement strand, we call them reverse complements. Note that we normally allow some errors in the repeats, as long as the number of errors is below some threshold. More precisely, those repeats should be called approximate repeats. Figure 5.1 shows examples of these types of repeats.

It has been noted that a large part of many genomes consists of repetitive sequences. Repetitive sequences play an important role in evolution and may serve as fundamental regulatory elements in the genome.

Tandem repeats, which are consecutive approximately repeated nucleotide sequences, constitute about 10% or more of the human genome [Benson 1999]. Trinucleotide

tandem repeat copies have been characterized in a number of human diseases, such as fragile-X mental retardation [Verkerk et al. 1991], Huntington's disease [Huntington's Disease Collaborative Research Group, 1993], spinal and bulbar muscular atrophy [Spada et al. 1991] and Friedreich's ataxia [Campuzano et al., 1996]. Tandem repeats may not only be involved in certain diseases, but may play a certain pivotal role in gene regulation, such as interacting with transcription factors, altering the chromatin structure or acting as protein binding sites [Hamada et al. 1984; Pardue et al. 1987; Yee et al. 1991; Richards et al. 1993; Lu et al. 1993].

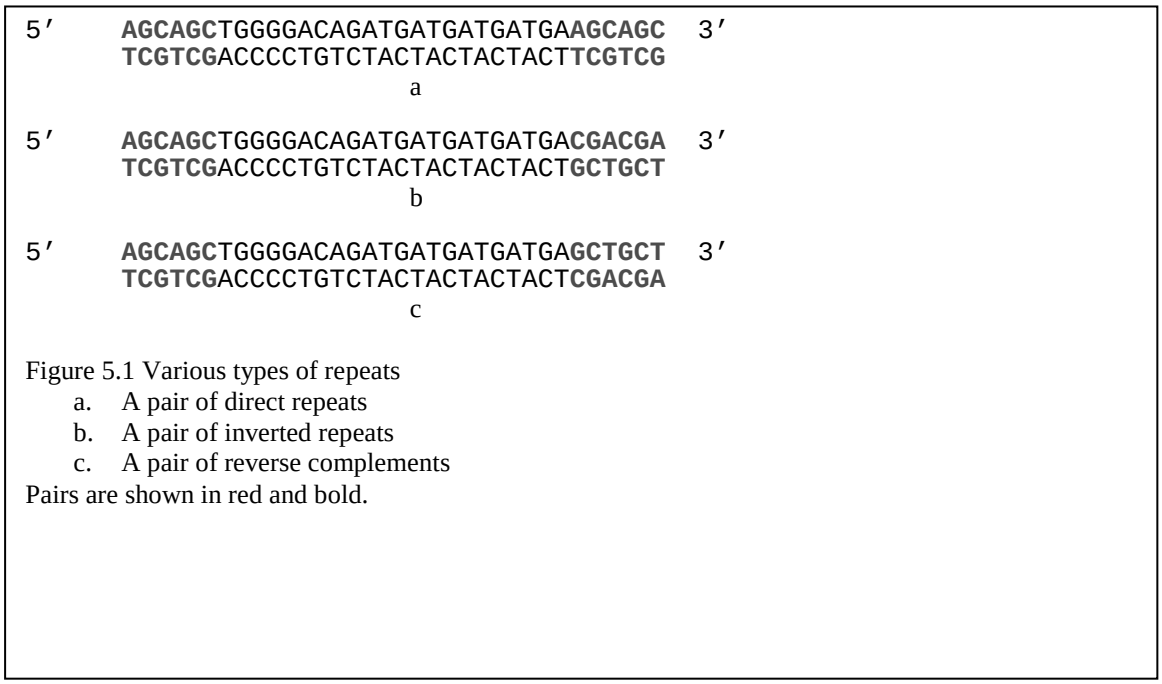


Figure 5.1 Various types of repeats
a. A pair of direct repeats
b. A pair of inverted repeats
c. A pair of reverse complements
Pairs are shown in red and bold.

Dispersed repeats appear mostly in transposable elements. Transposable elements can be grouped into three types: Class II Transposons, Miniature Inverted-repeats Transposable Elements (MITEs) (Class III) and Retrotransposons (Class I).

Class II transposons consist of DNA sequences that move from place to place within a genome. At the ends of a class II transposon, there is a pair of inverted repeats. After the transposon is inserted into host DNA sequences, a pair of direct repeats will flank the transposon [Transposons: Mobile DNA. 2004]. Figure 5.2 shows such a process.

In rice, approximately 26% of the genome sequences are derived from Transposable Elements (TEs), of which more than 70% are Miniature Inverted Repeat Transposable Elements (MITEs) [Jiang et al. 2004]. MITEs consist of hundreds of nucleotides flanked by a pair of reverse complements. The reverse complements consists of tens of nucleotides. Figure 5.3 shows an example of MITE.

Among the eukaryotic transposable elements, retrotransposons are the most abundant and widespread in the genome. There are two types of retrotransposons: the long terminal repeats (LTRs) and the non-LTR retrotransposons [Kumar and Bennetzen, 1999]. LTR retrotransposons contain direct long terminal repeats that have a size ranging from a few hundred nucleotides to over 5 kb [Kumar and Bennetzen, 1999]. Figure 5.4 shows the generic structure of an LTR retrotransposon element.

Complex biological patterns such as the repeat elements, Miniature Inverted Repeat Transposable Elements (MITEs), and the long terminal repeats (LTRs) have been a traditional focus in the research community. However, the enormous amount of sequence data hinders researchers in manually discovering them. The need for efficient computational algorithms to identify and locate those patterns has emerged in the

research community. A number of computational methods have thus been developed to help discover each of the above categories of patterns.

STRING is a heuristic approach for finding tandem repeats in DNA sequences [Parisi, 2003]. It uses a heuristic method to find all possible tandem repeats in DNA sequences. Two heuristic criteria are adopted in this algorithm. First of all, in order to reduce the regions where searching for tandem repeats is conducted, instead of studying the whole

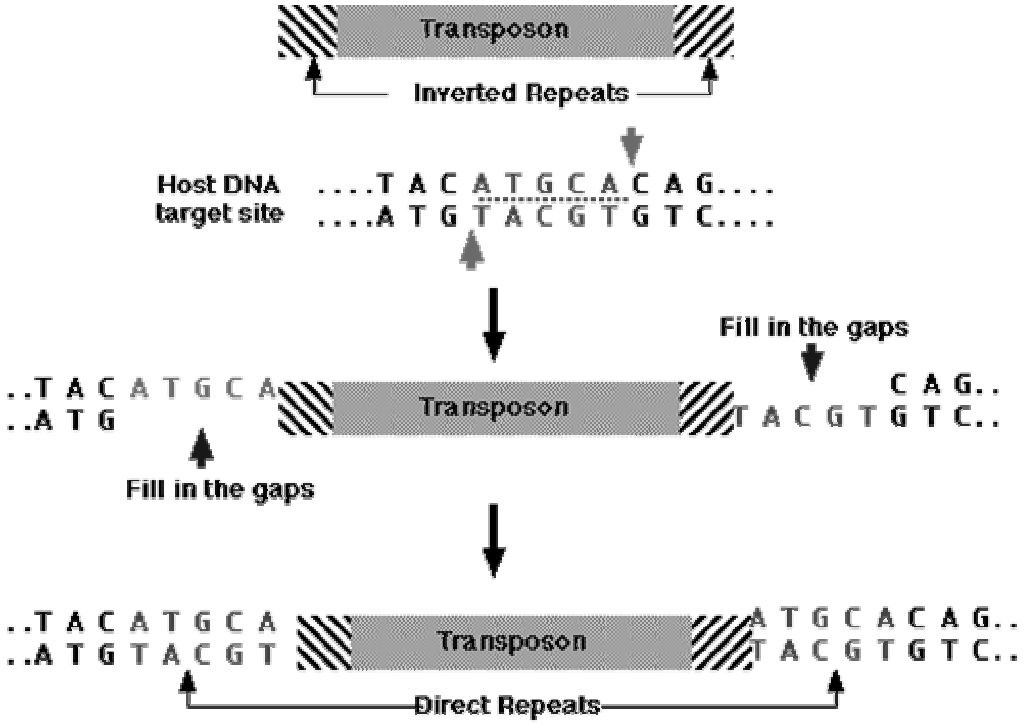


Figure 5.2. Process of how a class II Transposons moves into the host DNA, quoted from [Transposons: Mobile DNA. 2004].

sequence, only those regions that are considered to be more promising to contain tandem repeats according to an autoalignment procedure are examined. Note that autoalignment

5' GGAACCCTTTAAGGG..~400 nt..CCCTTAAAGGGTTCC 3'
3' CCTTGGGAAATTCCC..~400 nt.. GGGAATTTCCCAAGG 5'

Figure 5.3 Example of a MITE (Miniature Inverted Repeat Transposable Element)

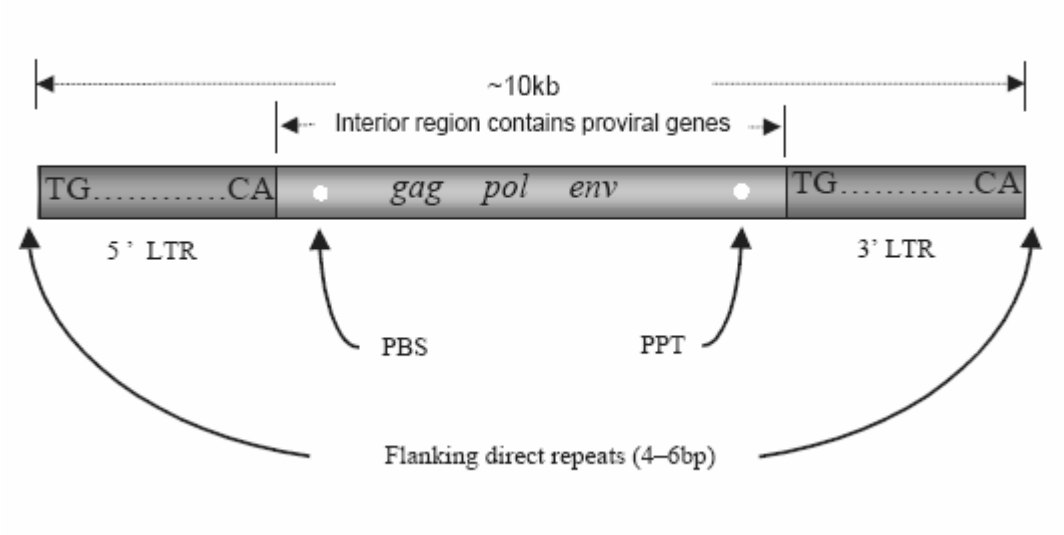


Figure 5.4 Generic structure of an LTR retrotransposon. Picture is taken from [McCarthy, et al. 2003]

is a procedure to search local alignments of a sequence with itself [De Fonzo et al., 1998]. Secondly, instead of studying all possible consensus tandem repeats, only those that are considered to be more promising ones (in a way inspired by the autoalignment procedure) are selected. This criterion will reduce the number of consensus tandem repeats being focused on in the search process.

The program ***mreps*** also aims at efficient and flexible detection of tandem repeats in DNA [Kolpakov et al. 2003] This program is able to identify all tandem repeats in the

genome with no limitation on the size of the repeated pattern. It can output tandem repeats with all possible pattern sizes. The algorithm first utilizes an efficient combinatorial algorithm [Kolpakov et al. 1999, 2001] to find all repetitive structures of a certain kind in a given sequence. This creates a set of raw repeat sequences that undergo further processing by a series of heuristic treatments: trimming the left and right edges of each repeat, computing the best period and merging for each repeat, filtering out statistically expected repeats and merging repeats with the same period p overlapping by at least $2p$.

TROLL, Tandem Repeat Occurrence Locator, [Castelo, 2002] is designed to find Simple Sequence Repeats (SSRs), which are those repetitive short nucleotide sequences with length less than six and that are fairly well conserved. It requires the user to provide a motif list and finds all occurrences of patterns from the motif list. TROLL is based on the *Aho Corasick* Algorithm (ACA) [Aho and Corasick, 1975], which was designed to find all occurrences from a list of patterns in a text. The ACA keeps track of a failure link while encountering a partial match in the text. It uses the failure link to continue the search and avoid re-examining characters in the text. TROLL implements the ACA to find pre-selected patterns in the text. It also records the tandem repeats such that the SSRs can be located.

Other programs such as EQUICKTANDEM from the EMBOSS package [Rice et al. 2000] and Tandem Repeats Finder [Benson 1999] have also been reported to facilitate finding tandem repeats. And in [Hauth et al. 2002], an algorithm was developed to find

two specific kinds of tandem repeats: variable length tandem repeats (VLTRs) and multi-period tandem repeats (MPTRs). Note that a variable length tandem repeat (VLTR) is a simple nested tandem repeat in which the copy number for some pattern is variable rather than constant, while a multi-period tandem repeat (MPTR) contains the nested concatenation of two or more *i*-similar patterns [Hauth et al. 2002], which are patterns that share *i* same characters.

REPEATMASKER [Smit et al. 2003] is a program developed to screen DNA sequences for interspersed repeats and low complexity DNA sequences. The program utilizes the program cross_match to perform sequence comparisons. The cross_match program is an efficient implementation of the Smith-Waterman-Gotoh algorithm with some enhancements.

RECON [Bao et al. 2002] is a program for the *de novo* identification and classification of direct repeat sequence families for sequenced genomes. It uses multiple alignment information to identify the boundaries of each copy of the repeats and to classify different repeat element families. RECON has been tested on the human genome to identify and group known transposable elements.

The program LTR_STRUC [McCarthy et al. 2003] is designed to automatically locate Long Terminal Repeat (LTR) retrotransposons from genome databases by searching for the structural features that exist in LTR retrotransposons. This program differs from

previous methods that were based on the sequence's similarity to previously identified LTR retrotransposons.

Although it seems there already are many programs developed to address the problem of locating complex patterns in sequences, we can still see that each existing program has been designed to locate a particular type of pattern, perhaps in a particular genome. What is lacking is a general tool that is able to facilitate locating all types of complex patterns in nucleotide sequences. Our tool, namely the Interactive Pattern Search Tool (IPST), is specifically designed to address this problem.

IPST utilizes a hashtable of n-mers for storage and fast retrieval of various patterns as well as direct and inverted repeats in DNA sequences. It is implemented in an interactive way such that users can load their sequences, input the features that they are looking for, and retrieve the output both through a graphical user interface and in the widely accepted GFF format [Sanger Center: GFF 2003]. IPST supports pattern search and finding various complex patterns not only over a single sequence but over multiple sequences. This feature enables IPST to find complex patterns at the genome level.

We have applied our tool to search for miniature inverted repeat transposable elements (MITEs) and for long terminal repeat (LTR) retrotransposon elements in rice sequences. In addition, as a test of the time and memory efficiency of our hashtable based algorithm in pattern search (approximate string match), we compared our hashtable based algorithm with an implementation of IPST based on a suffix tree approach.

5.2 Systems and methods

Algorithm and implementation

IPST uses a hashtable to facilitate searching the sequences. A hashtable is a data structure that maps a record to a key for efficient storage and retrieval of the data. IPST utilizes this data structure to facilitate finding patterns and repeats in the sequences.

Building the hashtable

Users first must provide the names of the sequence files. Three formats are accepted: FASTA, GCG and STADEN. After reading in the input sequences, IPST builds a hashtable in which the key values are polynucleotide subsequences. For a key of length 6, the maximum number of keys possible is 4^6. The longer and more varied the sequence is, the closer to this maximum the number of actual keys will be. Associated with each key is a value record that contains the position of the polynucleotide in the sequence and the index of the sequence in the set of multiple sequences being considered. This hashtable contains information for all of the sequences that users input. The process of hashtable-building is performed along each input sequence, and there will be at most (n-s+1) records for each sequence, where s is the length of the key. Since there may be more than one record corresponding to each key, these records are stored in a linked list. So the actual value corresponding to each key is a linked list of records. In practice, this process

is the major time-consuming process in the IPST algorithmic part, but this is compensated for by the quick access and retrieval of the patterns in the sequences.

A flexibility that IPST provides is that if users are only interested in part of the sequences, they can set delimiters for each input sequence before building the hashtable. This can save substantial computational time if only portions of the sequences are of interest to users.

Operations supported by IPST

IPST currently supports four major search operations (pattern search or approximate string matching, finding direct repeats, finding inverted repeats, and finding start and stop codons) as well as combinations of these operations. The steps for carrying out these operations are as follows.

1. Pattern search

Users enter the subsequence they want to search for as well as the number of mismatches they allow. IPST will locate those subsequences and display them on the graphical user interface with each pattern on a separate line. A textual output containing the result in GFF format [Sanger Center: GFF 2003] can be saved by the users by clicking the “report” button in the left panel.

2. Direct repeats

Users can find all pairs of direct repeats with specified minimum and maximum lengths, minimum and maximum spacing, and minimum percentage of repeat similarity within each sequence. Those pairs of direct repeats satisfying the criteria will be displayed on the graphical user interface. The textual results may be saved into a GFF-format [Sanger Center: GFF 2003] file.

3. Inverted repeats

Similar to the direct repeats, users can find all pairs of inverted repeats with specified minimum and maximum lengths, minimum and maximum spacing, and minimum percentage of repeat similarity within each sequence. Those pairs of inverted repeats satisfying the criteria will be displayed on the graphical user interface. The textual results can also be saved into a GFF-formatted [Sanger Center: GFF 2003] file.

4. Reverse complements

Given the specified minimum and maximum lengths, minimum and maximum spacing, and minimum percentage of repeat similarity, IPST can display the pairs of reverse complement on the graphical user interface and provide the GFF-format textual output.

5. Start and stop codons

Locate and display the positions of all the start and stop codons on both forward and reverse strands for each input sequence. An output containing the textual result in GFF format can also be created.

6. Combination of operations

The current implementation of IPST gives users the option of combining the pattern search operation with the operations of finding direct repeats, inverted repeats, or reverse complements. In the parameter setting panel for finding direct repeats, inverted repeats and reverse complements, users have the option to specify the pattern that needs to appear within the sequences that contain the repeats, and the minimum similarity for the pattern.

Figure 5.5 shows a snapshot of the graphical user interface of IPST with the results of a pattern search and direct repeats graphically displayed.

Underlying data structures and the procedure for each IPST-supported operation

There are nine major data structures involved in the IPST program: Hashtable, a Vector of SeqInfo objects, class SeqInfo, class DrawInfo_PS, DrawInfo_DR, DrawInfo_IR, DrawInfo_START, DrawInfo_STOP and class PolyNuclInfo.

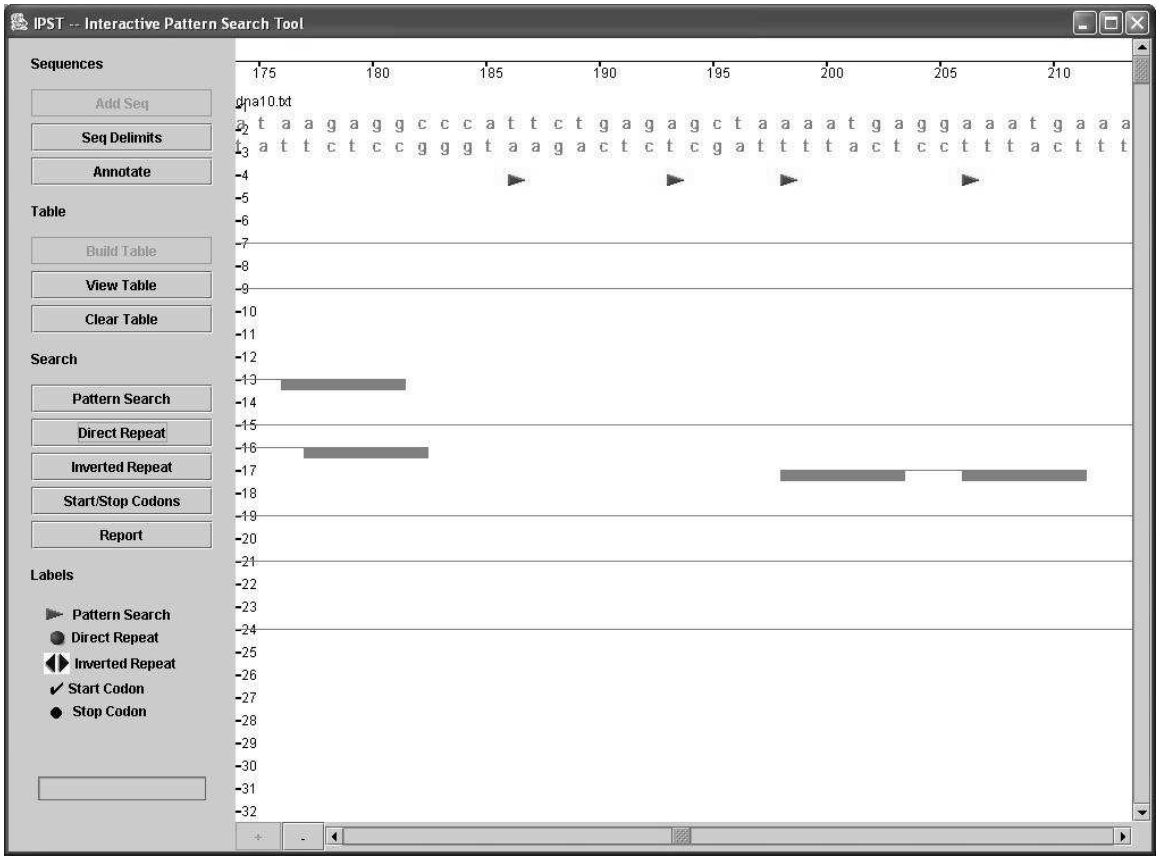


Figure 5.5. Snapshot of the IPST graphical user interface (with the results of a pattern search and direct repeats graphically displayed). The red arrows indicate the positions where the patterns were found during the pattern search process. The filled rectangles indicate the positions where each direct repeat is located. The lines connect pairs of direct repeats.

The class SeqInfo object contains the input sequence and its name, the delimiters of the sequence to be analyzed, and a series of DrawInfo classes (DrawInfo_PS, DrawInfo_DR, etc.) for each feature type, which contain the information about where to draw and what to draw on the canvas. Each sequence is associated with a SeqInfo object. The class PolyNuclInfo contains the position of the polynucleotide on the sequence and the index of the sequence in the vector that contains the SeqInfo objects. The Hashtable is used as

the central data structure for storing the sequence information. There are two Hashtables, one for the forward sequence and the other for the reverse sequence.

The relationship among all of the underlying data structures for IPST is shown in figure 5.6. A more detailed description about these data structures is available in figure 5.7.

With the hashtable and all other data structures established, the processes for performing the IPST operations are straightforward.

The pattern search process begins by searching the hashtable with the provided polynucleotide sequence as the key. If the length of input polynucleotide sequence is less than the default key length, the input polynucleotide will be extended by enumeration to every possible polynucleotide with the default length. Then each of these extended polynucleotides will be searched for against the hashtable. For example, given a key length of 6, and the pattern ATG, all 64 hexamer string beginning with ATG would be searched in the hashtable. In the case that the length of input polynucleotide sequence is greater than the default key length (6), the hashtable is searched with the first 6 nucleotides as the key. Once the list of PolyNuclInfo objects is retrieved, IPST will extend the first 6 nucleotides from each position along the sequence to check if the input polynucleotide sequence exists.

If a certain number of mismatches (users provided the minimum similarity percentage and a ceiling function is used to convert the calculated floating number of mismatches into an integer) are allowed for the pattern search, all possible polynucleotides with the

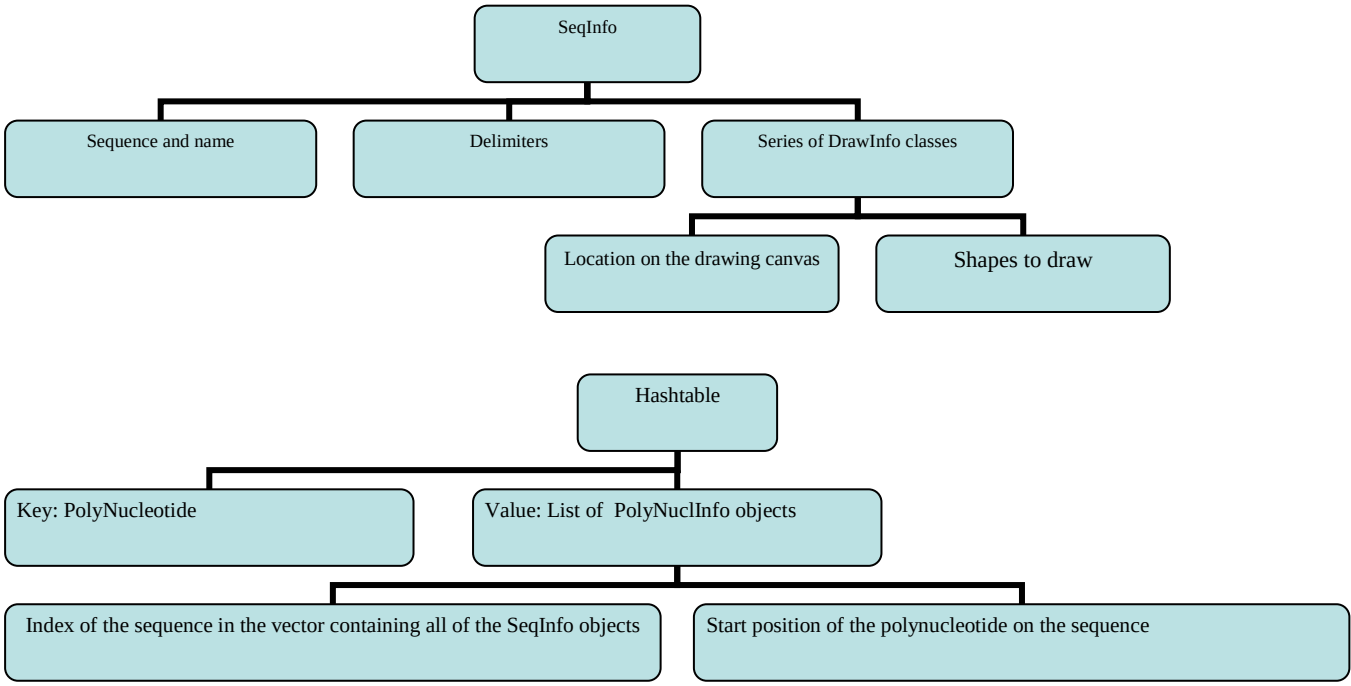


Figure 5.6: Relationships among IPST underlying data structures

length of the hashtable's key will be generated and then lists of PolyNuclInfo objects will be retrieved, extended along the sequence, and examined to determine whether the similarity between the extended polynucleotide and the pattern is above the similarity threshold. Retrieving the lists of PolyNuclInfo objects is fast and the main time-consuming step is the enumeration process. In order to reduce the number of enumerations, we adopted an assumption that the mismatch occurs in the first s (key length) nucleotides with the same probability as in all nucleotides of the whole pattern. Under this assumption, the number of enumerations is significantly reduced. For example, for a hashtable with key size of six and the common minimum similarity of 70% for the

	Key	Value
Data type	String	LinkedList
Content	Every possible polynucleotide in all sequences with the default number of nucleotides	Contains a list of PolyNuclInfo objects

Data structure of the Hashtable

	Element 1	Element 2
Data type	Int	Int
Content	The starting position of the polynucleotide on the sequence	The index of the sequence in the SeqInfoVector

Data structure of the class PolyNuclInfo

	Element 1	Element 2	Element 3	Element 4	Element 5 -- 9
Data type	String	int	int	String	Vector
Content	The sequence	The starting position of the sequence to display	The ending position of the sequence to display	The name of the sequence	Contains Objects of DrawInfo_PS, DrawInfo_DR, DrawInfo_IR, DrawInfo_START, and DrawInfo_STOP, respetively

Data structure of the class SeqInfo

	Element 1
Data type	Int
Content	The position on the sequence where objects should be drawn

Data structure of the class DrawInfo_PS

	Element 1	Element 2
Data type	Int	int
Content	The position on the sequence where objects should be drawn	Index of the DR

Data structure of the class DrawInfo_DR

	Element 1	Element 2
Data type	Int	int
Content	The position on the sequence where objects should be drawn	Index of the IR

Data structure of the class DrawInfo_IR

	Element 1	Element 2
Data type	Int	int
Content	The position on the sequence where objects should be drawn	Index of the SSIR

Data structure of the class DrawInfo_SameStrandIR

	Element 1
Data type	Int
Content	The position on the sequence where objects should be drawn

Data structure of the class DrawInfo_START

	Element 1
Data type	Int
Content	The position on the sequence where objects should be drawn

Data structure of the class DrawInfo_STOP

Figure 5.7. Underlying data structures of IPST

pattern search, the number of allowed mismatches is ceil((1-70%)*6) = 2, and the number of enumerations is only $(6*5)*4^2=480$. The number of enumerations will be only 24 given a minimum similarity of 80%. If the pattern is shorter than the size of the key, the pattern will be extended to meet the length of the key by adding enumerated nucleotides at each of the position for additional nucleotides. If mismatch is allowed, enumerations of the pattern will be performed prior to the extending process.

The procedure for finding the pairs of direct repeats or inverted repeats is very similar. The basic idea is to first retrieve the list of PolyNuclInfo objects by searching the hashtable with the first s-nucleotide sequence, where s is the default length of the hashtable's key, and then extend the polynucleotide sequence along the whole input sequence to see if there is a match. If users request a combination of pattern search and finding direct (inverted) repeats or reverse complement, a pattern search will be performed and only those pairs of repeats that contain the specified pattern will be located. For direct and inverted repeats and reverse complements, if a pair of repeats overlaps the other, covers the other pair, or if they have the same delimiters but one pair of repeats is longer than the other one, the shorter pairs of repeats will be removed in both cases.

Figure 5.8 shows the procedure for searching the pairs of direct repeats.

The procedures for locating the start and stop codons are very similar to those for the pattern search.

For each key in the PolyNuclTable

1. If pattern search is also requested, perform the pattern search operation.
2. Get the LinkedLists containing the PolyNuclInfo(s) for the key.
3. If the above list is not null and has more than two elements
   For each pair of PolyNuclInfos in the list
      If both of these PolyNuclInfo(s) are from the same sequence
         do
            Extend the polynucleotide along the sequence
         while
            The spacing between this pair of direct repeats is still within the specified range AND
            The similarity of these two repeats remains above the threshold
         If there is match
            (if the pattern search operation is also performed) if the pair of repeats contains the pattern
               Record the positions and sequence index into the DrawInfo_DR object.
4. Eliminate overlapped repeats.

Figure 5.8. the procedure for searching for pairs of direct repeats

The graphical user interface is designed for users to interact with the program in a friendly manner. Users can input their sequences, click the corresponding buttons to let the program build the table, set their criteria for pattern searches, direct and inverted repeats, and view the results displayed in the graphical user interface. Furthermore, IPST can provide a GFF-formatted textual report of the results [Sanger Center: GFF 2003].

The Java-programming language

IPST is implemented using the Java-programming language [Java, 2004]. The version that we used to compile and run the Java codes is 1.4.2.

Test sequences on which IPST was applied

Rice sequence 1

This sequence of size around 340kb was downloaded from NCBI (http://www.ncbi.nlm.nih.gov/). The GenBank accession number of this sequence is AF172282. It is the adh1-adh2 region of the rice *Oryza sativa*.

Rice sequence 2

This sequence of size around 160 kb was downloaded from NCBI (http://www.ncbi.nlm.nih.gov/) as well. The GenBank accession number of this sequence is AC020666. It is a BAC genomic sequence on the chromosome 10 of the rice *O. sativa*.

Sequence for time-memory testing

Seven sequences with sizes ranging from 10 kb, 20 kb, 50 kb, …, to 500kb are created by cut and paste (start from the beginning) of a concatenated series of *Zea mays* BAC (Bacterial Artificial Chromosome) clones from the GenBank. Refer to http://www.cs.uga.edu/~eileen/IPST to view the file containing the concatenated series of *Zea mays* BAC sequences.

Time-memory testing of hashtable based IPST versus suffix tree based IPST

IPST utilizes a hashtable based approach to perform approximate string matching and find the direct and inverted repeats. In order to test the time and memory performance, we also implemented the pattern match (approximate string matching) procedure using a suffix tree based approach, as implemented in a set of Java codes written by [Dorohonceanu et al. 2000]. Since this Java-implementation of the suffix tree data structure and string search is a general suffix tree implementation designed for representing multiple sequences and searching strings on a multi-sequence context, it is well suited for our test purposes and acts as a good comparison of these approaches.

We performed two sets of tests. First, we tested the time and memory usage of these two versions of IPST for a certain pattern search with test sequences of various sizes. In the second set, we tested the time and memory usage of these two versions of IPST for searching patterns with various lengths (3 bp, 6 bp, 12 bp and 24 bp) on the same sequence (the 100 kb test sequence).

Tests were executed on a PC with Intel Pentium 4 CPU 3.00 GHz and 512 MB of RAM, running Microsoft windows XP Professional Version 2002.

IPST is implemented using the Java-programming language [Java 2004]. Testing the memory usage in the Java environment is problematic because the Java-programming language uses a garbage collector to automatically deal with the memory usage while the

program is running. We adopted the approach for memory usage testing from [Roubtsov 2002], which uses three techniques to accurately measure the memory usage. These three techniques are as follows:

1. Use Runtime.totalMemory()-Runtime.freeMemory() to calculate the used heap size instead of a single call to Runtime.freeMemory().
2. Execute many Runtime.gc() calls and request object finalizers to stabilize the perceived heap size.
3. Ignore heap space consumed by the first class instance.

5.3 Results and Discussions

Apply IPST to find MITEs (Miniature Inverted-repeats Transposable Elements) in sequence 1

As shown in figure 5.3, MITEs (Miniature Inverted-repeats Transposable Elements) contain a pair of reverse complements at the ends of the sequence. To apply IPST to locate this element, we need to find the reverse complements in the sequence.

IPST provides a panel of parameter settings for users to define the types of reverse complements they wish to find. Users can interact with IPST by providing different parameter settings and producing different outputs. Having considered the property of the

MITEs and interactively explored a number of parameter settings, we chose the following parameter settings for the operation of finding reverse complements in this sequence.

Minimum length of reverse complement: 10
Maximum length of reverse complement: 100
Minimum spacing between the pair of reverse complements: 200
Maximum spacing between the pair of reverse complements: 500
Minimum identity between the pair of reverse complements: 0.95

We did not choose the option of combining the pattern search.

It took 1.343 seconds time and 28,669,984 bytes memory for IPST to build the hashtable. Then it took 78.266 seconds and 89864 bytes memory to find those potential MITEs with our computer (Intel Pentium 4 CPU 3.00 GHz and 512 MB of RAM, running Microsoft windows XP Professional Version 2002). Our analysis has revealed a total of 256 potential MITEs in this sequence. Figure 5.9 and 5.10 show part of the GFF-format textual report and the graphical user interface (GUI) for finding the MITEs in sequence 1, respectively.

We have also examined the annotation of sequence 1 from GenBank, where a total of 33 MITEs have been annotated. Between the 256 MITEs found by IPST and those 33 MITEs from the GenBank annotation, 19 share the same locations to a large extent.

```
## IPST -- Interactive Pattern Search Tool, designed to find repeats in DNA
sequences
## 2004-07-15 14:05:28

<seqname>   <source><feature>            <start> <end>  <score><strand><frame><length>[comments]

## Reverse
complements

6979318.fastaIPST     reverse_complement2308362312401     .     .     10     pair 1
6979318.fastaIPST     reverse_complement1092921096851     .     .     10     pair 2
6979318.fastaIPST     reverse_complement1247871250211     .     .     10     pair 3
6979318.fastaIPST     reverse_complement1700181703041     .     .     10     pair 4
6979318.fastaIPST     reverse_complement1673871678321     .     .     11     pair 5
6979318.fastaIPST     reverse_complement1757821762231     .     .     11     pair 6
6979318.fastaIPST     reverse_complement3247663251031     .     .     10     pair 7
6979318.fastaIPST     reverse_complement4398  4735  1     .     .     10     pair 8
6979318.fastaIPST     reverse_complement12374 12685 1     .     .     10     pair 9
6979318.fastaIPST     reverse_complement1570621575791     .     .     11     pair 10
```

Figure 5.9 Part of the GFF-format textual report for finding miniature inverted repeat transposable elements (MITEs) sequence 1

Detailed examination on those 14 missed MITEs needs to be performed to analyze why IPST failed to find these elements. Then users can experiment with different parameter settings for the reverse complements that will be able to lead to the identification of those 14 MITEs. In addition, the fact that the number of MITEs found by IPST is much larger

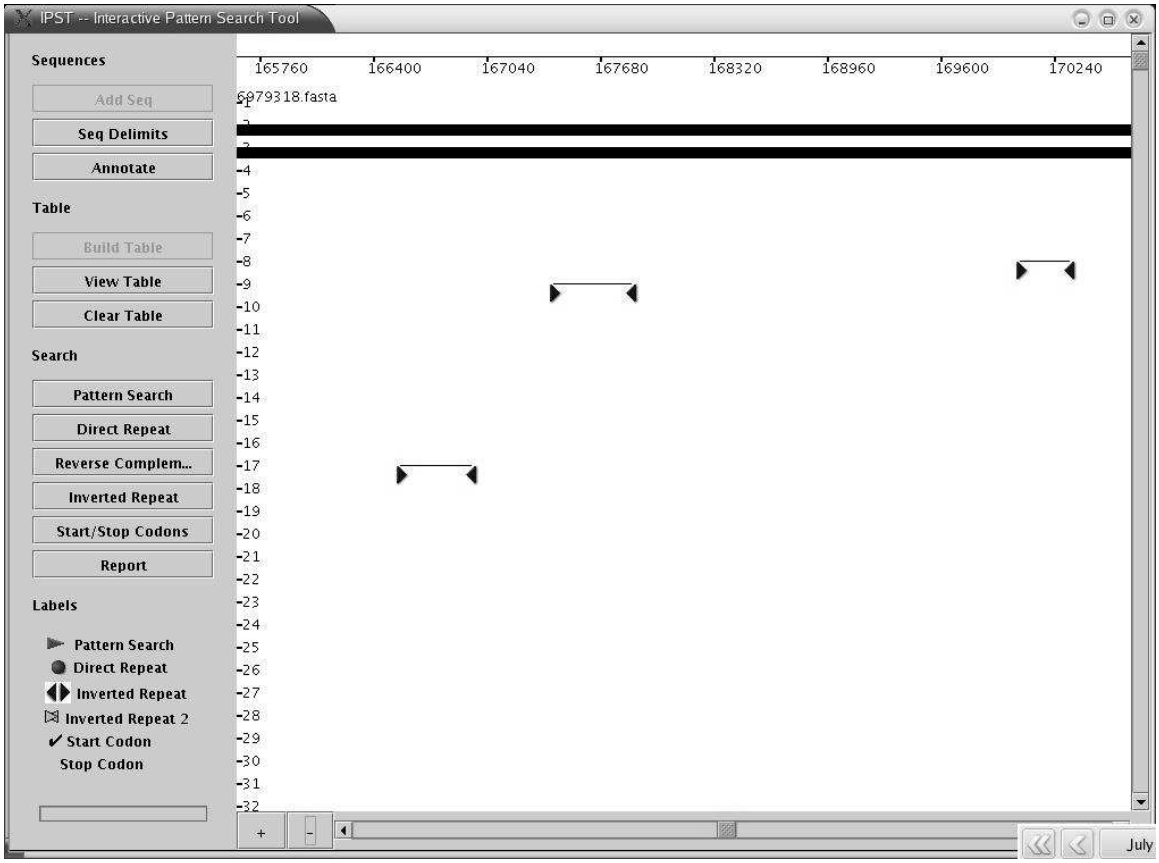


Figure 5.10 Snapshot of some MITEs Miniature Inverted-repeats Transposable Elements) from the rice sequence found by IPST.

than the number of annotated MITEs in the GenBank suggests other operations, such as pattern search, can be combined with finding the reverse complements to exclude those "false" MITEs. In order to do that, further study needs to be done to find consensus sequences in MITEs. On the other hand, there may be some MITEs that have not been annotated and need further study for confirmation. Many of these processes can be done through user interaction with the IPST program.

Apply IPST to find Long Terminal Repeat retrotransposons in sequence 2

Since Long Terminal Repeat (LTR) retrotransposons contain a pair of direct repeats at the ends of the sequence, the operation of finding direct repeats needs to be performed in IPST in order to find this element. In addition, the interior region of the LTR retrotransposons contains several genes such as *env, gag, and pol*, as shown in figure 5.3. A combination of finding the direct repeats with pattern search will be very effective to locate the element.

These genes often have consensus amino acid sequences. Ideally it will be very useful if IPST can also be applied to pattern search for amino acid sequences, which will be effective to find the LTR retrotransposons. However, the current implementation has yet to include amino acid sequences as the inputs. So we simply combine the operations of finding the pattern “ATG” (start codon) with finding the direct repeats to locate the LTR retrotransposons in this sequence

Similar to finding reverse complements, IPST provides a panel for parameter settings of the direct repeats. After experimenting with a variety of parameter settings with IPST, we used the following parameter settings for the operation of finding direct repeats:

Minimum direct repeat length: 100
Maximum direct repeat length: 100000
Minimum spacing between Direct Repeats: 1000

Maximum spacing between Direct Repeats: 1000000

Minimum identity of Direct Repeats: 0.95

We clicked the “and” checkbox to perform the pattern search as well. We searched for “atg” with an identity value of “1.0”.

It took 0.75 seconds time 13831160 bytes memory for IPST to build the hashtable. Then it took 5.719 seconds time and 80656 bytes memory to find those potential Long Terminal Repeat (LTR) retrotransposons with our computer (Intel Pentium 4 CPU 3.00 GHz and 512 MB of RAM, running Microsoft windows XP Professional Version 2002). Our analysis has revealed a total of 13 potential LTR retrotransposons that contain both the direct repeats and the pattern “ATG”.

Figure 5.11 and figure 5.12 show the GFF-format textual report and the graphical user interface (GUI) for finding the Long Terminal Repeat (LTR) retrotransposons in sequence 2, respectively.

Time-memory tests

In order to measure the time and memory efficiency of our hashtable based approach for string match, we compared both the theoretical time and memory complexity and the empirical running time and memory consumed while applying IPST on pattern search (approximate string matching).

IPST -- Interactive Pattern Search Tool, designed to find repeats in DNA sequences

2004-07-15 14:43:48

<seqname>	<source>	<feature>	<start>	<end>	<score>	<strand>	<frame>	<length>	[comments]
	repeats		pattern						
## Find Direct that		contain the	"atg"						
20198551.fasta	IPST	direct_repeat	77337	93548	0.950980392.		.	102	pair 1;pattern at 78208
20198551.fasta	IPST	direct_repeat	141397	155407	0.952702703.		.	148	pair 2;pattern at 142925
20198551.fasta	IPST	direct_repeat	141288	146090	0.953703704.		.	108	pair 3;pattern at 142925
20198551.fasta	IPST	direct_repeat	141297	146107	0.956896552.		.	116	pair 4;pattern at 142925
20198551.fasta	IPST	direct_repeat	36495	47607	0.953703704.		.	108	pair 5;pattern at 37628
20198551.fasta	IPST	direct_repeat	141353	155405	0.952631579.		.	190	pair 6;pattern at 142925
20198551.fasta	IPST	direct_repeat	77313	93546	0.951612903.		.	124	pair 7;pattern at 78208
20198551.fasta	IPST	direct_repeat	78098	94297	0.956896552.		.	116	pair 8;pattern at 78208
20198551.fasta	IPST	direct_repeat	65731	77170	0.95025729 .		.	583	pair 9;pattern at 67077
20198551.fasta	IPST	direct_repeat	78774	94839	0.95049505 .		.	101	pair 10;pattern at 81316
20198551.fasta	IPST	direct_repeat	77803	94097	0.952631579.		.	190	pair 11;pattern at 78208
20198551.fasta	IPST	direct_repeat	78089	94280	0.953703704.		.	108	pair 12;pattern at 78208
20198551.fasta	IPST	direct_repeat	78074	94270	0.955752212.		.	113	pair 13;pattern at 78208

Figure 5.11 GFF-format textual report for finding the Long Terminal Repeat (LTR) retrotransposons in sequence 2

To build the hashtable, the algorithm employed in IPST needs to go through each nucleotide in the input sequence and apply the hashing function to the s-mer beginning at that position. Since applying the hashing function on each polynucleotide can be considered to take constant time and there are a total of (n-s+1) polynucleotides along the sequence (where n is the length of the sequence and s is the length of the key for the hashtable), the time complexity for building the hashtable is $O(n)$.

The space needed for building the hashtable consists of the space needed by all keys in the hashtable and by the values, which are linked lists of PolyNuclInfo objects. Both a key and a PolyNuclInfo object occupy constant space. The maximum number of all possible keys is 4^s, (we have four different nucleotides: a, g, c and t) where s is the size of the key. The maximum possible number of PolyNuclInfo objects in a linked list is (n-s+1). So the space complexity for the hashtable approach is $O(\max(n, 4^s)$. Assuming that s is small and constant, the space complexity is $O(n)$.

To perform the pattern search with mismatch allowed, if the pattern is no shorter than the hashtable key, our hashtable approach first enumerates all possible key values based on the percentage of similarity for the whole pattern under our assumption that the mismatch occurs in the first s (key length) nucleotides with the same probability as in all nucleotides of the whole pattern. This process takes $O(4^k * C(s,k))$ (C(s,k) is the math expression for "s choose k") time where k is the number of allowed mismatches and 4 is the number of different nucleotides. Then lists of PolyNuclInfo objects will be retrieved, extended along the sequence, and examined to determine whether the similarity between the extended polynucleotide and the pattern is above the similarity threshold. Applying the hashing function and retrieving the values can be considered to take constant time. The time needed for extending the polynucleotide and checking for the similarity is $O(m-s)$, where m is the length of the pattern. So the time complexity for the pattern search in our hashtable approach is $O((m-s) * 4^k * C(s,k))$ if the pattern is larger than the hashtable key. If the pattern has the same size as the key, the complexity will be $O(4^k C(s,k))$. If the

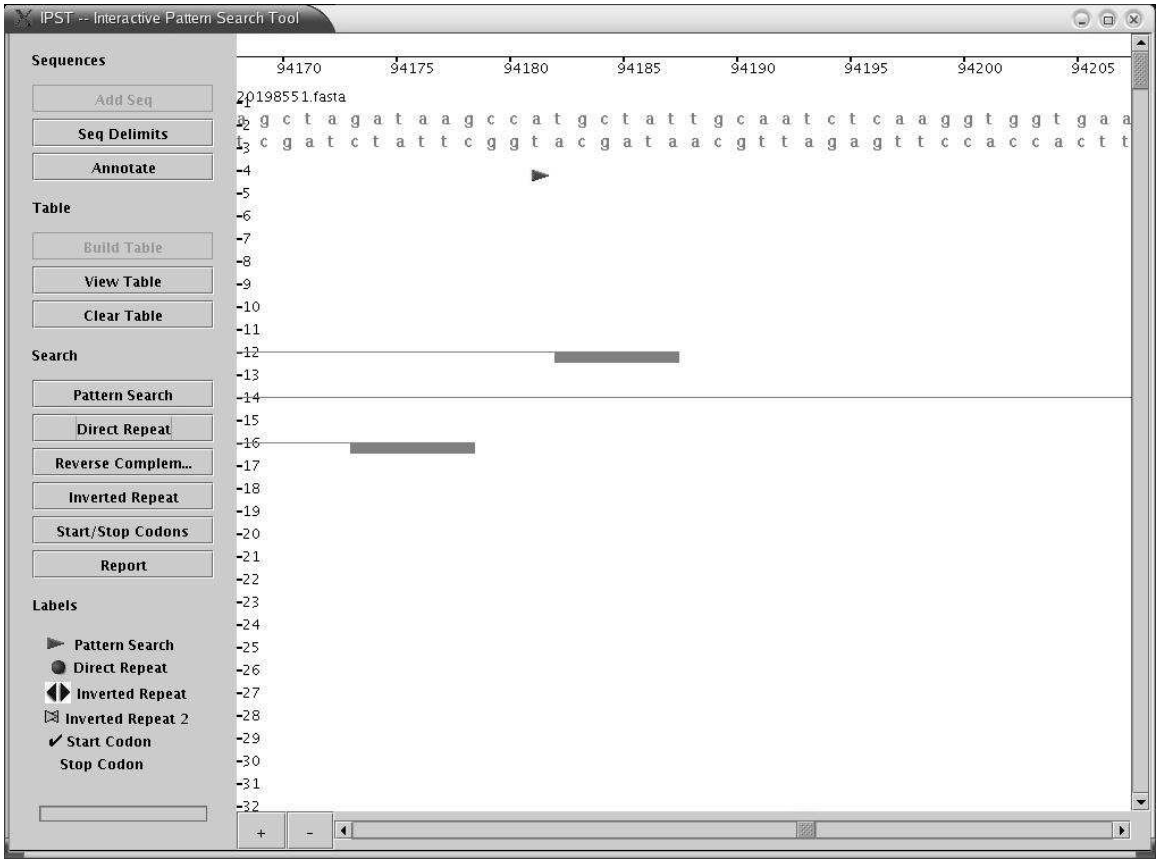


Figure 5.12 Snapshot of part of the LTR retrotransposons from the rice sequence found by IPST.

pattern is shorter than the key, the pattern will be extended to meet the length of the key by adding enumerated nucleotides at each of the position for additional nucleotides. When mismatch is allowed, the number of enumerations will be $4^k * C(s,k) * 4^{s-m}$. So the time complexity for pattern search while the pattern is shorter than the key size is $O(4^{k+s-m} * C(s,k))$.

As for the space complexity of the hashtable approach for the pattern search, storing the hashtable takes $O(n)$, as discussed above for the space complexity of building the

hashtable. Storing the pattern takes space $O(m)$ The space complexity needed for pattern search in our hashtable approach is $O(m + n)$.

Table 5.1 shows the time and memory complexity comparison between our hashtable based approach and the suffix tree approach. This table indicates that these two approaches share the same time complexity for preprocessing and the same space complexity for pattern search. In our hashtable approach, both the space complexity for preprocessing and the time complexity for pattern search depend on the size of the hashtable key. If the key of the hashtable is relatively small, the two approaches will have similar space complexity for preprocessing. Otherwise, our hashtable approach will consume more space. As for the time complexity for pattern search in our hashtable approach, it varies with different relationships between the size of the pattern and that of the hashtable key. When the pattern has the same size as the key or has a larger size than the key, our hashtable approach takes less time than the suffix tree approach. The hashtable approach is less efficient if the pattern is shorter than the key.

A Java-implementation of the suffix tree and search from [Dorohonceanu et al. 2000] was adopted into the IPST program using the same interface to perform the time-measure on the test sequences.

Figure 5.13 shows the time-memory test comparison between the suffix tree approach and our hashtable approach on 6 sequences ranging from 10kb to 500kb. The hashtable

	Preprocessing		Patten Search	
	Time complexity	Space complexity	Time complexity	Space complexity
Suffix tree	$O(n)^a$	$\Theta(n\log\|\Sigma\|)^a$	$O(km)^a$	$O(n+m)^a$
Hashtable	$O(n)$	$O(\max(n, 4^s)$; $O(n)$ assuming s is small and constant	$O((m-s) * 4^k * C(s,k))$ if $m > s$ $O(4^k C(s,k))$ if $m = s$ $O(4^{k+s-m} * C(s,k))$ if $m < s$	$O(n+m)$

Table 5.1: Time and memory complexity comparison between our hashtable based approach and the suffix tree approach.

n: text size; m: pattern size; k: number of mismatches allowed; s: length of key for building the hashtable

a: cited from [Gusfield, 1997]

was built with the default key size of 6. The string “agattcgaacgt” was searched with 2 mismatches allowed.

Figure 5.13a shows that the preprocessing time needed for the hashtable approach is significantly less than what is needed for the suffix tree approach. While the size of the input sequence increases, our hashtable approach becomes even more time efficient for the preprocessing. While table 5.1 shows these two approaches have the same time complexity, our experimental results on the test sequences indicate that the implementation of the suffix tree approach from [Dorohonceanu et al. 2000] has not achieved the same efficiency as our hashtable implementation.

The message from figure 5.13b confirms with our space complexity comparison between these two approaches: our hashtable consumes more space during the preprocessing stage than the suffix tree approach.

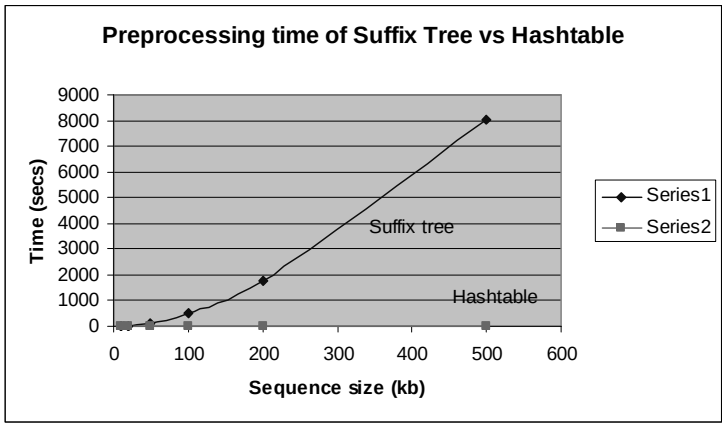
Figure 5.13c shows the comparison of time efficiency for the two approaches on pattern search. The time needed for these two approaches on the test sequences was in small amount. While the two approaches take similar time for pattern search against input sequences of size up to 100kb, our hashtable approach is two times faster than the suffix tree approach. Figure 5.13d indicates that the memory needed for our hashtable approach during the pattern search process is always at least 40% less than what is needed for the suffix tree approach in out tests.

Based on figure 5.13, the preprocess time needed for the hashtable approach is much less than what is needed in the suffix tree approach and the time and memory efficiencies of our hashtable approach are always superior than that of the suffix tree approach on pattern search (or approximate string matching). The only drawback of the hashtable approach is that it needs more memory in the preprocessing phase.

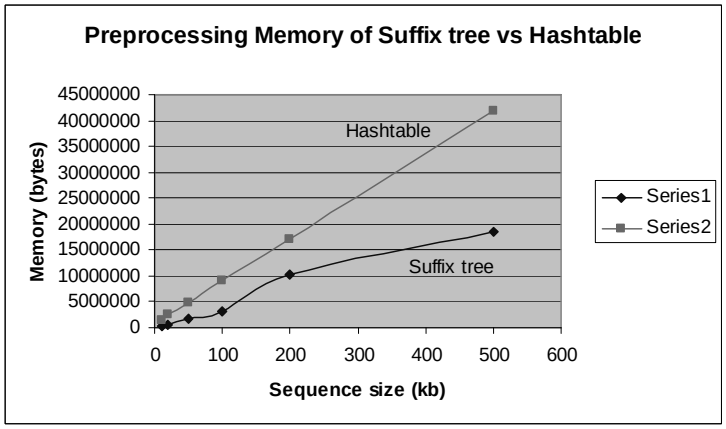
Figure 5.14 shows the time-memory comparisons for the suffix tree based and hashtable based IPST implementation for approximate string matching on patterns with various lengths. Figure 5.14a indicates that our hashtable approach is up to three times more efficient than the suffix tree approach whenever the pattern is longer than the key. The

benefit of our hashtable approach increases while the pattern is longer. Figure 5.14b shows that while the size of the pattern increases, the amount of memory to be spent on the pattern search becomes very close between these two approaches. As we have discussed in the previous complexity analysis, our hashtable approach is not time-efficient when the size of the pattern is less than that of the hashtable's key. This is indicated in figure 5.14a where it takes much more time to find the pattern "atg" for the hashtable approach than for the suffix tree approach.

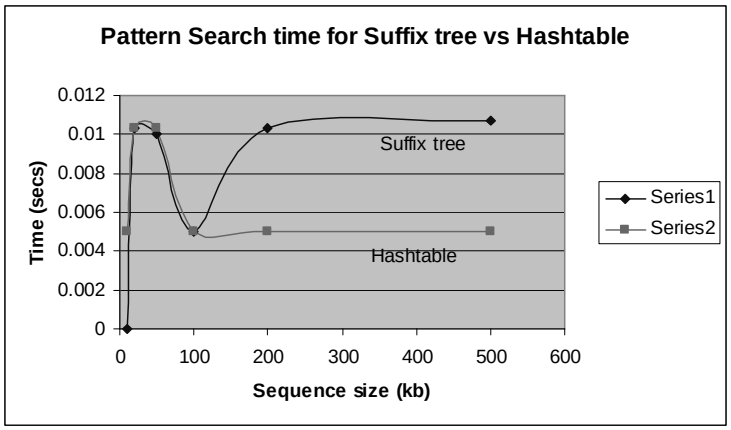
Approximate string matching, as a basic computational problem, has acted as a fundamental basis for many other problems in computational biology, such as primer design, sequence alignment, homology study etc. Due to its importance,
a large number of approaches have been developed for the approximate string matching problem aiming to improve the time and space efficiency. These approaches include the dynamic programming approach based on edit distances [Gusfield, 1997], some algorithms based on automata [Wu, et al. 1996], bit-parallelism algorithms [Wu et al., 1992], filtering algorithms based on cutting unmatched contents [Chang and Marr, 1994], the suffix tree approach [Ukkonen, 1995] and other heuristic methods. Table 5.2 shows our survey of current approximate string matching algorithms and their time and space complexity. The heuristic method employed in our IPST program for the approximate string match (as well as for finding the direct and inverted repeats) utilizes a preprocessed hashtable to expedite the searching process. The empirical time and space comparison between our approach and the suffix tree approach showed that our approach can achieve the same or better time efficiency in approximate string matching as the suffix tree


Preprocessing time of Suffix Tree vs Hashtable
Time (secs)
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
100
200
300
400
500
600
Sequence size (kb)
Suffix tree
Hashtable
Series1
Series2

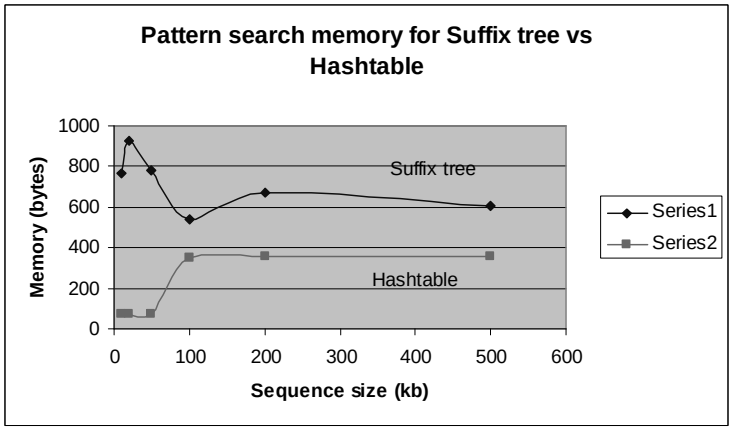
a


Preprocessing Memory of Suffix tree vs Hashtable
Memory (bytes)
45000000
40000000
35000000
30000000
25000000
20000000
15000000
10000000
5000000
0
0
100
200
300
400
500
600
Sequence size (kb)
Hashtable
Suffix tree
Series1
Series2

b



c



d

Figure 5.13. Time-memory test comparisons between the suffix tree approach and the hashtable based approach for approximate string matching on six test sequences with various lengths.

i. For suffix tree, preprocessing means the process of building the suffix tree in the suffix tree approach and the process of building the hashtable in the hashtable approach

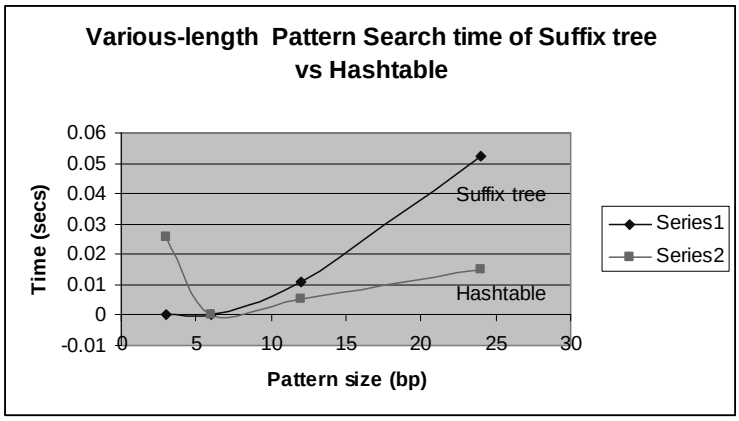
ii. For all pattern seaches, “agattcgaacgt” was searched in the test sequences with 2 mismatch allowed. All time and memory test results are averaged on three tests.

approach while requiring much less preprocessing time. From table 5.2 we can see that the suffix tree approach is the most time-efficient among all of the surveyed approximate string matching approaches. Thus, our hashtable based approach for approximate string matching should also compare well against the other methods.

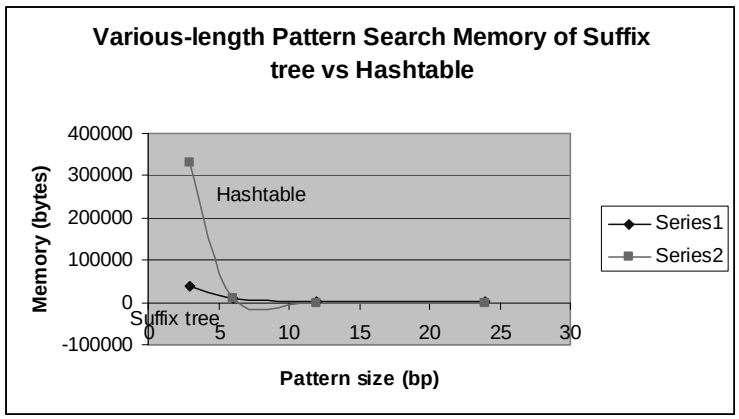
5.4 Summary and future work

An Interactive Pattern Search Tool (IPST) has been developed to facilitate finding various complex patterns in biological sequences in an interactive manner. Current functions of this tool include pattern search, finding direct repeats, inverted repeats and reverse complements and a combination of the above operations. IPST has been demonstrated to find LTR (Long Terminal Repeat) retrotransposons and MITEs (Miniature Inverted-repeats Transposable Elements) in rice sequences. The time and memory efficiency of the hashtable based heuristic approach for approximate string match implemented in this tool has been compared with another implementation of IPST using the suffix tree approach.

Future work includes developing more combinational operations (such as OR, NOT, XOR) in IPST and making the combinational operation more versatile and user-interactive. In some situations, users need to find some complex pattern over many different input sequences. A nice feature for IPST to include is to be able to generate the codes (in our case, Java classes) that support the operations or the set of combinational operations to find a particular complex pattern. In this way, users simply need to use the



a



b

Figure 5.14. Time-memory tests for suffix tree based and hashtable based IPST for approximate string matching on patterns with various lengths. All results are averaged on three tests.

i. All patterns are searched for against the 100 kb test sequence.

ii. For the pattern of length 3, “atg” was searched for with 0 mismatch allowed (or 0.8 identity)

iii. For the pattern of length 6, “agcatc” was searched for with 1 mismatch allowed (or 0.8 identity)

iv. For the pattern of length 12, “agattcgaacgt” was searched for with 2 mismatches allowed (or 0.8 identity)

v. For the pattern of length 24, “gaacttgcaggtatcacttgcatc” was searched for with 2 mismatches allowed (or 0.9 identity)

Approach name	Space complexity	Time complexity	Source
The dynamic programming approach	O(m)	O(mn)	Navarro, 2001
The cut-off heuristic algorithm	O(m)	O(kn) on average [b]	Ukkonen, 1985
Column partitioning algorithm	O(mσ) [a]	O(kn/sqrt(σ)) on average [a]	Chang and Lampe, 1992
Wu, Manber and Myers' algorithm based on automata	O(n) [a]	O(kn/logn) on average [a]	Wu, et al. 1996
Wu and Manber's Bit-parallelism algorithm	[c]	O(k ⌈m/w⌉ n) [a]	Wu et al., 1992
Yates and Navarro's Bit-parallelism algorithm	[c]	O(⌈k(m-k)/w⌉ n) on worst case; O(⌈k^2/w⌉ n) on average case [a]	Baeza-Yates et al. 1999
Myers' approach on parallelizing the dynamic programming matrix	[c]	O (⌈m/w⌉ n) for worst case; O (⌈k/w⌉ n) on average	Myers, 1999
Chang and Marr's Filtering algorithm	O (m^t)	O (n(k+logσm)/m) on average (σ is the base of the log; t is some constant which depends on σ)	Chang and Marr, 1994
Suffix tree approach	Θ(mlog|Σ|) space for building the tree; O(n+m) for approximate string matching	O(m) time for building the tree; O(km) for approximate string matching	Ukkonen, 1995

Table 5.2. A survey of the time and space complexities of the currently-developed algorithms for approximate string matching (pattern search). Note that k is the number of mismatches allowed. m is the length of the pattern to search for and n is the length of the text

a: proved in [Navarro, 2001] b: proved in [Chang and Lampe, 1992; Baeza-Yates and Navarro 1999]

c: data unavailable

generated codes to find the complex pattern that they are interested in. In addition, it will be very useful for this tool to be able to perform all of these functions on amino acid sequences.

The motivation and application of IPST discussed so far are mainly based on exploring biological sequences to find interesting patterns. If we adapt the alphabet into a user-supplied character set, we can explore both protein sequences and complex patterns in arbitrary texts.

ACKNOWLEDGMENTS

I would like to thank the following people who have used tested this tool on their experimental data and provided me with a lot of great feedbacks and suggestions: Drs. Russell Malmberg, Ning Jiang, Mary Bedell, and Mr. Renyi Liu. I would also like to thank Drs. Russell Malmberg and Liming Cai for their ideas that led to the development of this project.

References

Aho, A.V. and Corasick, M.J. (1975) Efficient string matching: an aid to bibliographic search. *Communication of the ACM*, 18, 333-340.

Baeza-Yates, R. and Navarro, G. 1999. Faster approximate string matching. *Algorithmica* 23, 2, 127-158. Preliminary versions in Proceedings of CPM '96(LNCS, vol. 1075, 1996) and in Proceedings of WSP'96, Carleton Univ. Press, 1996.

Bao, Z. and Eddy, S. Automated De Novo Identification of Repeat Sequence Families in Sequenced Genomes. *Genome Research*. 12:1269-1276, 2002.

Benson,G. (1999) Tandem repeats finder: a program to analyse DNA sequences. *Nucleic Acids Res.*, **27**, 573–580.

Campuzano,V., Montermini,L., Molto,M.D., Pianese,L., Cossee,M., Cavalcanti, F., Monros, E., Rodius, F., Duclos, F., Monticelli, A., Zara, F., Canizares, J., Koutnikova, H., Bidichandani, S., Gellera, C., Brice, A., Trouillas, P., Michele, G., Filla, A., Frutos, R., Palau, F., Patel, P., Donato, S., Mandel, J., Cocozza, S., Koenig, M., Pandolfo, M. Friedreich's Ataxia: Autosomal Recessive Disease Caused by an Intronic GAA Triplet Repeat Expansion. (1996) *Science*, **271**, 1423-1427.

Castelo,A.T., Martins,W. and Gao,G.R. TROLL—tandem repeat occurrence locator. Bioinformatics, 18, 634–636. 2002.

Chang, W. and Lampe, J. 1992. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching (CPM '92)*. LNCS, vol.644, Springer-Verlag, Berlin, 172-181.

Chang, W. and Marr, T. 1994. Approximate string matching and local similarity. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching (CPM '94)*. LNCS, vol. 807, Springer-Verlag, Berlin, 259-273.

De Fonzo,V., Bersani,E., Aluffi-Pentini,F., Castrignanò,T. and Parisi,V. (1998) Are only repeated triplets guilty? *J. Theor. Biol.*, **194**, 125–142.

Dorohonceanu, B. and Nevill-Manning, C. A Practical Suffix-Tree Implementation for String Searches. *Dr. Dobb's Journal*. July. 133-140. 2000.

Hamada,H., Seidman,M., Howard,B. and Gorman,C. Enhanced Gene Expression by the Poly(dT-dG) · Poly(dC-dA) Sequence. (1984) *Mol. Cell. Biol*., **4**, 2622-2630.

Hauth,A.M. and Joseph,D.A. Beyond tandem repeats: complex pattern structures and distant regions of similarity. Bioinformatics, 18, S31–S37. 2002.

Huntington's Disease Collaborative Research Group. A Novel Gene Containing a Trinucleotide Repeat That Is Expanded and Unstable on Huntington's Disease Chromosomes. (1993) *Cell*, **72**, 971-983.

Java. (2004) http://java.sun.com/.

Jiang N, Feschotte C, Zhang X, and Wessler S R **(2004)**. Using rice to understand the origin and amplification of miniature inverted repeat transposable elements (MITEs). *Curr. Opin. Plant Biol.* 7:115-119.

Kolpakov,R. and Kucherov,G. Finding maximal repetitions in a word in linear time. In Proceedings of the 1999 Symposium on Foundations of Computer Science, New York (USA), IEEE Computer Society, pp. 596–604. 1999.

Kolpakov,R. and Kucherov,G. Finding approximate repetitions under Hamming distance. In Meyer auf der Heide,F. (ed.), 9th European Symposium on Algorithms (ESA 2001), Aarhus, Denmark, Volume 2161 of Lecture Notes in Computer Science, pp. 170–181. 2001.

Kolpakov, R. Bana1, G. and Kucherov, G. *mreps*: efficient and flexible detection of tandem repeats in DNA. *Nucleic Acids Research*. Vol. 31, No. 13: 3672–3678. 2003.

Kumar, A., and J. L. Bennetzen (1999) Plant retrotransposons. *Ann. Rev. Genet.* 33:479-532.

Spada,A.R. La, Wilson,E., Lubahn,D., Harding,A. and Fischbeck,K. Androgen receptor gene mutations in X-linked spinal and bulbar muscular atrophy. (1991) *Nature*, **352**, 77-79.

Myers, G. 1999. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J.ACM* 46, 3, 395-415. Earlier version in *Proceedings of CPM'98*(LNCS, vol. 1448).

Navarro, Gonzalo, A Guided Tour to Approximate String matching. *ACM Computing Surveys*, Vol. 33, No. 1, March 2001.

Parisi, V., De Fonzo, Valeria and Aluffi-Pentini, Filippo. STRING: finding tandem repeats in DNA sequences. *Bioinformatics.* Vol. 19 no. 14:1733–1738.2003.

Sanger Center: GFF (2003) GFF (General Feature Format) Specifications Document. http://www.sanger.ac.uk/Software/formats/GFF/GFF_Spec.shtml

Transposons: Mobile DNA. (2004). http://users.rcn.com/jkimball.ma.ultranet/BiologyPages/T/Transposons.html

Lu,Q., Wallrath,L., Granok,H. and Elgin,S. $(CT)_n \cdot (GA)_n$ Repeats and Heat Shock Elements Have Distinct Roles in Chromatin Structure and Transcriptional Activation of the *Drosophila hsp26* Gene. (1993) *Mol. Cell. Biol*., **13**, 2802-2814.

Pardue,M., Lowenhaupt,K., Rich,A. and Nordheim,A. $(dC\text{-}dA)_n \cdot (dG\text{-}dT)_n$ sequences have evolutionarily conserved chromosomal locations in Drosophila with implications for roles in chromosome structure and function. (1987) *EMBO J*., **6**, 1781-1789.

Rice,P., Longden,I. and Bleasby,A. EMBOSS: The european molecular biology open software suite. Trends Genet., 16, 276–277. 2000.

Richards,R., Holman,K., Yu,S. and Southerland,G. Fragile X syndrome unstable element, $p(CCG)_n$, and other simple tandem repeat sequences are binding sites for specific nuclear proteins. (1993) *Hum. Mol. Genet*., **2**, 1429-1435.

Roubtsov, V. Java Tip 130: Do you know your data size? (2002) http://www.javaworld.com/javaworld/javatips/jw-javatip130.html

Smit, AFA and Green, P. RepeatMasker. http://repeatmasker.org . 2003.

Ukkonen, E. 1985. Finding approximate patterns in strings. *J. Algor*. 6, 132-137.

Ukkonen, E. On-line construction of suffix-trees. Algorithmica, 14:249-60, 1995.

Wu, S. and Manber, U. 1992. Fast text searching allowing errors. *Commun. ACM* 35, 10, 83-91.

Wu, S., Manber, U., and Myers, E. 1996. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica* 15, 1, 50-67. Preliminary version as Tech. Rep. TR29-36, Computer Science Dept., Univ. of Arizona, 1992.

Verkerk,A., Pieretti,M., Sutcliffe,J. Fu,Y., Kuhl,D., Pizzuti,A., Reiner,O., Richards,S., Victoria,M., Zhang,F., Eussen,B., van Ommen,G., Blonden,A., Riggins,G., Chastain,J., Kunst,C., Galjaard,H., Caskey,C., Nelson,D., Oostra,B. and Warren,S. Identification of a Gene (*FMR-1*) Containing a CGG Repeat Coincident with a Breakpoint Cluster Region Exhibiting Length Variation in Fragile X Syndrome. (1991) *Cell*, **65**, 905-914.

Yee,H., Wong,A., Sande, J.H.van de, and Rattner,J. Identification of novel single-stranded d$(TC)_n$ binding proteins in several mammalian species. (1991) *Nucleic Acids Res.*, **19**, 949-953.

Chapter 6

Conclusions and future work

In this thesis work, we have evaluated five computer programs on their ability to locate coding regions and to predict gene structure in the organism *N. crassa*. We have also designed and implemented a tool for automatic evaluation of various gene-finding programs. This tool can be applied in various genome projects to facilitate choosing the right gene-finding program for a specific organism. It can also be used to assess the performance of newly-developed gene-finding programs.

In addition, an Interactive Pattern Search Tool (IPST) has been developed to facilitate finding various complex patterns in biological sequences in an interactive manner. Current functions of this tool include pattern search, finding direct repeats, inverted repeats and reverse complements and any combination of the above operations. IPST has been demonstrated to find LTR (Long Terminal Repeat) retrotransposons and MITEs (Miniature Inverted-repeats Transposable Elements) in rice sequences. The time and memory efficiency of the hashtable based heuristic approach for approximate string match implemented in this tool has been compared with another implementation of IPST using the suffix tree approach. Future work of this IPST project includes developing more functions in the tool, providing more combinational operations among those implemented functions, and applying the tool to find other complex patterns in both nucleotide and amino acid sequences.

References

Aho, A.V. and Corasick, M.J. Efficient string matching: an aid to bibliographic search. *Communication of the ACM*, 18, 333-340, 1975.

Arlazarov, V., Dinic, E., Konrod, M. and Faradzev, I. On economic construction of the transitive closure of a directed graph. *Sov. Math. Dokl. 11*, 1209-1210, 1975. Original in Russian in *Dokl. Akad. Nauk SSSR* 194, 1970.

Bao, Z. and Eddy, S. Automated De Novo Identification of Repeat Sequence Families in Sequenced Genomes. *Genome Research*. 12:1269-1276, 2002.

Baeza-Yates, R. and Navarro, G. 1998. New and faster filters for multiple approximate string matching. Tech. Rep. TR/DCC-98-10, Dept. of Computer Science, University of Chile. Random Struct. Algor. To appear.
ftp://ftp.dcc.ptuchile.cl/pub/users/gnavarro/multi.ps.gz

Baeza-Yates, R. and Navarro, G. Faster approximate string matching. *Algorithmica* 23, 2, 127-158, 1999. Preliminary versions in Proceedings of CPM '96(LNCS, vol. 1075, 1996) and in Proceedings of WSP'96, Carleton Univ. Press, 1996.

Bayat, Ardeshir. Science, medicine, and the future: Bioinformatics. *British Medical Journal*. 324: 1018-1022, 2002.

Benson,G. Tandem repeats finder: a program to analyse DNA sequences. *Nucleic Acids Res.*, **27**, 573–580, 1999.

Bioinformatics Factsheet. 2001.
http://www.ncbi.nlm.nih.gov/About/primer/bioinformatics.html

Burset,M. Guigo,R. Evaluation of gene structure prediction programs. *Genomics*, 34:353-367, 1996.

Castelo,A.T., Martins,W. and Gao,G.R. TROLL—tandem repeat occurrence locator. Bioinformatics, 18, 634–636, 2002.

Chang, W. and Lampe, J. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching (CPM '92)*. LNCS, vol.644, Springer-Verlag, Berlin, 172-181, 1992.

Chang, W. and Marr, T. Approximate string matching and local similarity. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching (CPM '94)*. LNCS, vol. 807, Springer-Verlag, Berlin, 259-273, 1994.

Charras, Christian and Lecroq, Thierry. http://www-igm.univ-mlv.fr/~lecroq/string/. 1997

De Fonzo,V., Bersani,E., Aluffi-Pentini,F., Castrignanò,T. and Parisi,V. Are only repeated triplets guilty? *J. Theor. Biol.*, **194**, 125–142, 1998.

Delgrangey, O. and Rivals, E. STAR: an algorithm to Search for Tandem Approximate Repeats. *Bioinformatics*. June 4. (Advance Access from online). 2004.

Dorohonceanu, B. and Nevill-Manning, C. A Practical Suffix-Tree Implementation for String Searches. *Dr. Dobb's Journal*. July. 133-140, 2000.

Feschotte, C, Jiang, N and Wessler, S R. Plant transposable elements: where genetics meets gemomics. Nature Reviews Genetics. 3: 329-341, 2002.

Fischetti, V. A., Landau, G. M., Sellers, P. H. & Schmidt, J. P. Identifying periodic occurrences of a template with applications to protein structure. *Inf Proc Letters*, **45**, 11.18, 1993.

Gusfield, Daniel, Algorithm on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge Press, 1997.

Hauth,A.M. and Joseph,D.A. Beyond tandem repeats: complex pattern structures and distant regions of similarity. Bioinformatics, 18, S31–S37. 2002.

Jokinen, P., Tarhio, J., and Ukkonen, E. A comparison of approximate string matching algorithms. *Software Practice Exper.* 26, 12, 1439-1458, 1996. Preliminary version in Tech. Rep. A-1991-7, Dept. of Computer Science, Univ. of Helsinki, 1991.

Knuth, D.E., Morris, J.H., Pratt, V.B. Fast pattern matching in strings. *SIAM J. Compt.*, 6:323-50, 1977.

Kolpakov,R. and Kucherov,G. Finding maximal repetitions in a word in linear time. In Proceedings of the 1999 Symposium on Foundations of Computer Science, New York (USA), IEEE Computer Society, pp. 596–604, 1999.

Kolpakov,R. and Kucherov,G. Finding approximate repetitions under Hamming distance. In Meyer auf der Heide,F. (ed.), 9th European Symposium on Algorithms (ESA 2001), Aarhus, Denmark, Volume 2161 of Lecture Notes in Computer Science, pp. 170–181, 2001.

Kolpakov, R. Bana1, G. and Kucherov, G. mreps: efficient and flexible detection of tandem repeats in DNA. *Nucleic Acids Research*. Vol. 31, No. 13: 3672–3678, 2003.

Kraemer E, Wang J, Guo J, Hopkins S, Arnold J. **An analysis of gene-finding programs for *Neurospora crassa*.** *Bioinformatics.* 7(10): 901-912, 2001.

Myers, G. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J.ACM* 46, 3, 395-415, 1999. Earlier version in *Proceedings of CPM'98*(LNCS, vol. 1448).

Navarro, G. and Raffinot, M. Fast and flexible string matching by combining bit-paralleslism and suffix automata. *ACM J. Exp. Algor*. 5, 4, 2000. Previous version in *Proceedings of CPM '98*. Lecture Notes in Computer Science, Springer-Verlag, New York.

Navarro, Gonzalo, A Guided Tour to Approximate String matching. *ACM Computing Surveys*, Vol. 33, No. 1, March 2001.

Parisi, V., De Fonzo, Valeria and Aluffi-Pentini, Filippo. STRING: finding tandem repeats in DNA sequences. *Bioinformatics.* Vol. 19 no. 14:1733–1738, 2003.

Rice,P., Longden,I. and Bleasby,A. EMBOSS: The european molecular biology open software suite. Trends Genet., 16, 276–277, 2000.

Smit, AFA and Green, P. RepeatMasker. http://repeatmasker.org . 2003.

Sutinen, E. and Tarhio, J. On using q-gram locations in approximate string matching. In Proceedings of the 3rd Annual European Symposium on Algorithms (ESA '95). LNCS, vol. 979, Springer-Verlag, Berlin, 327-340, 1995.

Takaoka, T. Approximate pattern matching with samples. *In Proceedings of ISAAC '94.* LNCS, vol. 834, Springer-Verlag, Berlin, 234-242, 1994.

Tarhio, J. and Ukkonen, E. Approximate Boyer-Moore string matching. *SIAM J. Comput.* 22, 2, 243-260, 1993. Preliminary version in *SWAT'90*(LNCS, vol. 447, 1990).

Ukkonen, E. Finding approximate patterns in strings. *J. Algor.* 6, 132-137, 1985.

Ukkonen, E. On-line construction of suffix-trees. *Algorithmica*, 14:249-60, 1995.

Wang J and Kraemer E. **GFPE: Gene-finding Program Evaluation** *Bioinformatics* 19(13): 1712-1713, 2003.

Wu, S. and Manber, U. Fast text searching allowing errors. *Commun. ACM* 35, 10, 83-91, 1992.

Wu, S., Manber, U., and Myers, E. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica* 15, 1, 50-67, 1996. Preliminary version as Tech. Rep. TR29-36, Computer Science Dept., Univ. of Arizona, 1992.