SIDE EFFECT MACHINES FOR SEQUENCE CLASSIFICATION

A Thesis

Presented to

The Faculty of Graduate Studies

of

The University of Guelph

by

ELIZABETH WARNER

In partial fulfilment of requirements

for the degree of

Master of Science

March, 2009

©Elizabeth Warner, 2009

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-52280-6
Our file Notre référence
ISBN: 978-0-494-52280-6

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

ABSTRACT

SIDE EFFECT MACHINES FOR SEQUENCE CLASSIFICATION

Elizabeth Warner

University of Guelph, 2009

Advisor:

Daniel Ashlock

Finite state machines are often used to efficiently recognize patterns in strings. The internal state structure of the machine is typically only of peripheral interest, appearing in algorithms only when the number of states is minimized in the interests of efficiency of execution or comparison. A *side effect machine* saves information about the internal transitions of the state machine. This record of internal state transitions forms a numerical feature set for the string run through the machine which allows it to be classified. Finite state machines are trained with an evolutionary algorithm to produce feature sets that are very easy for an unsupervised learning algorithm, *k*-means clustering, to learn. The system is demonstrated on a collection of synthetic and biological DNA data. Parameters are explored to characterize their effect on performance. Side effect machines are found to be effective at recognizing classes of DNA sequence data.

ACKNOWLEDGEMENTS

Many thanks to all the people who helped me with this thesis. In particular, thank you to my advisor, Daniel Ashlock, for providing the code for my simulations, all his advice, comments, and helpful academic and non-academic discussions. Thank you to the members of my advisory committee, Anna Lawniczak and Deborah Stacey, for their helpful comments and the members of my examination committee, Gary Umphrey and Gerarda Darlington. I'd like to give a shout-out to my office-mates in MacNaughton 526 (as well as those in neighboring offices); you are the only ones who truly experienced the day-in, day-out routine of research and writing with me. Thanks to my family and friends for their ongoing encouragement; and finally, a special thanks goes to Paul for making me laugh more than I ever thought was possible and giving me more support than I could have imagined.

Contents

1	Introduction	1
1.1	DNA	3
1.2	The Human Genome	4
1.3	Evolutionary Computation	4
1.4	Side Effect Machines	9
1.5	Evolving Side Effect Machines	10
2	Definitions and Theorems on Side Effect Machines	14
2.1	Tag Detector Side Effect Machine	14
2.2	GC-content Side Effect Machine	16
2.3	Substring Statistic Side Effect Machine	17
3	Experimental Design	24
3.1	Fitness Function	24
3.1.1	k -Means Clustering	25
3.1.2	Rand Index	27
3.2	Synthetic Data	29

3.2.1	Entropy Bounded Data	29
3.2.2	GC-Content Data	34
3.2.3	Stochastic Fitness Experiment with Motif Location Data	35
3.3	Biological Data	36
3.3.1	PCR Primer DNA	37
3.3.2	Mitochondrial DNA	41
3.3.3	Intron and Exon DNA	42
4	Results and Discussion	46
4.1	Results and Discussion for Synthetic Data	46
4.1.1	Entropy Bounded Experiments	46
4.1.2	GC-Content Experiments	62
4.1.3	Stochastic Fitness Experiments	70
4.2	Results and Discussion for Biological Data	92
4.2.1	PCR Primer DNA Experiments	92
4.2.2	Mitochondrial DNA Experiments	97
4.2.3	Intron and Exon DNA Experiments	101
5	Conclusion and Future Direction	106
5.1	Conclusions	106
5.2	Future Directions	107
5.2.1	Ignorance Side Effect Machines	107
5.2.2	k -Nearest Neighbor Algorithm	109
5.2.3	Contrasting Data Targets	110

A	Code	118
A.1	Side Effect Machine Evolver Code	118
A.2	Side Effect Machine Population Creator Code	125
A.3	Side Effect Machine Population Creator Header File	129
A.4	k -Means Clustering Code	130
A.5	k -Means Clustering Header File	132
A.6	Cross-validation Code	133
A.7	GC-Content Data Creator Code	137
A.8	Partial Fitness Evaluation Code	139
A.9	Ignorance Side Effect Machine Evolver Code	147
A.10	Ignorance Side Effect Machine Population Creator Code	153
A.11	Ignorance Side Effect Machine Population Creator Header File	157

List of Figures

1.1	Two-point crossover.	6
1.2	Adaptive crossover.	7
1.3	Example of a side effect machine.	10
1.4	Side effect machine representation.	11
1.5	Side effect machine diagram of representation shown in Figure 1.4.	11
1.6	Example of two-point crossover.	12
1.7	Two-point mutation on side effect machine from Figure 1.4.	13
2.1	Side effect machine used to detect the substrings G,G,T and G,A,T.	16
2.2	Side effect machine used to calculate GC-content of a DNA string.	17
2.3	Example of a line graph construction.	19
2.4	Line graph construction of the three smallest binary De Bruijn graphs. . . .	20
2.5	Rotating drum diagram of the 3-D binary line graph shown in Figure 2.4. . .	21
2.6	Substring statistics for $n = 2$	23
3.1	Flowchart demonstrating the side effect machine fitness function.	25
3.2	k -means clustering algorithm.	27
3.3	Rand index example.	28

3.4	Self-driving Markov automaton.	30
3.5	The polymerase chain reaction.	39
3.6	DNA to protein formation.	44
4.1	Parameter study results for data set 3.	47
4.2	Parameter study results for data set 4.	48
4.3	Parameter study results for data set 5.	49
4.4	95% confidence intervals and best-of-run fitness for data set 3.	51
4.5	95% confidence intervals and best-of-run fitness for data set 4.	52
4.6	95% confidence intervals and best-of-run fitness for data set 5.	53
4.7	Fitness scores by generation for data set 3, 12-states.	54
4.8	Fitness scores by generation for data set 3, 20-states.	55
4.9	Fitness scores by generation for data set 3, 28-states.	56
4.10	Fitness scores by generation for data set 4, 12-states.	57
4.11	Fitness scores by generation for data set 4, 20-states.	58
4.12	Fitness scores by generation for data set 4, 28-states.	59
4.13	Fitness scores by generation for data set 5, 12-states.	60
4.14	Fitness scores by generation for data set 5, 20-states.	61
4.15	Fitness scores by generation for data set 5, 28-states.	62
4.16	Mean fitness scores by generation for GC-content data.	63
4.17	95% confidence intervals and best-of-run fitness for GC-content data.	64
4.18	Cross-validation for GC-content data, 4-states, 1 mutation.	65
4.19	Cross-validation for GC-content data, 4-states, 2 mutations.	66

4.20	Cross-validation for GC-content data, 12-states, 1 mutation.	67
4.21	Cross-validation for GC-content data, 12-states, 2 mutations.	68
4.22	Cross-validation for GC-content data, 20-states, 1 mutation.	69
4.23	Cross-validation for GC-content data, 20-states, 2 mutations.	70
4.24	Mean fitness scores by generation for <i>full</i> data set.	71
4.25	95% confidence intervals and best-of-run fitness for <i>full</i> data set.	72
4.26	Mean fitness scores by generation for <i>partial</i> data set.	73
4.27	95% confidence intervals and best-of-run fitness for <i>partial</i> data set.	74
4.28	Cross-validation for <i>full</i> data set, 12-states, 1 mutation.	75
4.29	Cross-validation for <i>full</i> data set, 12-states, 2 mutations.	76
4.30	Cross-validation for <i>full</i> data set, 20-states, 1 mutation.	77
4.31	Cross-validation for <i>full</i> data set, 20-states, 2 mutations.	78
4.32	Cross-validation for <i>full</i> data set, 28-states, 1 mutation.	79
4.33	Cross-validation for <i>full</i> data set, 28-states, 2 mutations.	80
4.34	Cross-validation for <i>partial</i> data set, 12-states, 1 mutation.	81
4.35	Cross-validation for <i>partial</i> data set, 12-states, 2 mutations.	82
4.36	Cross-validation for <i>partial</i> data set, 20-states, 1 mutation.	83
4.37	Cross-validation for <i>partial</i> data set, 20-states, 2 mutations.	84
4.38	Cross-validation for <i>partial</i> data set, 28-states, 1 mutation.	85
4.39	Cross-validation for <i>partial</i> data set, 28-states, 2 mutations.	86
4.40	Cross-validation for <i>full</i> & <i>partial</i> data sets, 12-states, 1 mutation.	87
4.41	Cross-validation for <i>full</i> & <i>partial</i> data sets, 12-states, 2 mutations.	88
4.42	Cross-validation for <i>full</i> & <i>partial</i> data sets, 20-states, 1 mutation.	89

4.43	Cross-validation for <i>full & partial</i> data sets, 20-states, 2 mutations.	90
4.44	Cross-validation for <i>full & partial</i> data sets, 28-states, 1 mutation.	91
4.45	Cross-validation for <i>full & partial</i> data sets, 28-states, 2 mutations.	92
4.46	Mean fitness scores by generation for primer data, 12-states.	93
4.47	Mean fitness scores by generation for primer data, 16-states.	94
4.48	Mean fitness scores by generation for primer data, 20-states & 24-states. . .	95
4.49	95% confidence intervals and best-of-run fitness for primer data.	96
4.50	95% confidence intervals and best-of-run fitness for mt data, 1 mutation. . .	98
4.51	95% confidence intervals and best-of-run fitness for mt data, 2 mutations. . .	99
4.52	Mean fitness scores by generation for mitochondrial data, 1 mutation.	100
4.53	Mean fitness scores by generation for mitochondrial data, 2 mutations. . . .	101
4.54	95% confidence intervals and best-of-run fitness for in/ex data, 1 mutation. .	102
4.55	95% confidence intervals and best-of-run fitness for in/ex data, 2 mutations.	103
4.56	Mean fitness scores by generation for intron & exon data, 1 mutation.	104
4.57	Mean fitness scores by generation for intron & exon data, 2 mutations. . . .	105
5.1	Example of an ignorance side effect machine as a linear array.	108
5.2	k -nearest neighbor example.	109
5.3	Stem-loop structure.	111

Chapter 1

Introduction

This thesis introduces a novel technique for DNA sequence classification that utilizes evolutionary computation. This is an initial feasibility study of a new technique. It is hypothesized that training, via evolution, variable length character strings to fixed length numerical transducers yields good classification accuracy.

This thesis begins by surveying other techniques used to classify DNA. In [15], Hirsh and Noordewier use characterized DNA to form classifiers to accurately label regions in uncharacterized DNA. This particular study uses two popular inductive learning methods: a decision-tree learning algorithm called C4.5 rules, and a neural-network training algorithm called backpropagation [15]. In [28], sequence analysis software is used to show sequence diversity in protein-coding genes of bacteria. This sequence diversity is used to cluster ecologically diverse populations of bacteria [28]. In [14], Henrissat also makes use of sequence analysis software in classifying common enzymes called glycosyl hydrolases based on their amino acid sequence similarities. In [21] a new type of inductive learning method is used called compression-based induction. Compression-based induction is a text compression

technique that forms DNA models that are more suitable for sequence learning. It focuses on the relative locations of nucleotide bases and on the dependence of consecutive locations instead of more common sequence identification methods that depend on absolute locations and independence of consecutive nucleotide locations [21]. In [23] an expectation maximization algorithm is used to locate the binding sites of an *E. coli* promoter sequence. The expectation maximization algorithm is useful in this study because there are many variable features that make finding the binding sites of an *E. coli* promoter sequence difficult, see [23] for more detail. These features are then run through a neural-network for promoter recognition. In [36], two new techniques are used to classify DNA sequences. These techniques calculate scores based on differences from unclassified sequences to a set of classified sequences. In summary, non-laboratory techniques currently used to classify DNA sequences include the use of decision-trees, neural-networks, software for distinguishing sequence diversity or similarities, text compression, and the use of a score metric comparing unclassified sequences to classified sequences.

Recognition of functional roles of DNA regions plays an important role in the biological sciences, as the laboratory process to do so is a rather painstaking one [15]. Thus, non-laboratory methods for classification and identification of functional roles of DNA sequences are much sought after. Evolutionary computation has been proven in many studies to be an efficient technique for pattern searching large spaces of data and is used to optimize the parameters and/or structure of many types of classifiers [10].

1.1 DNA

Deoxyribonucleic acid (DNA) contains genetic information that defines the development and functioning of individual living organisms. It has two basic functions, it provides a blueprint for converting amino acids into proteins and provides a template for reproduction of somatic cells and gametes. The average adult human has approximately 100 trillion cells in which each cell contains identical DNA, only the expression of that code can be expressed differently in different types of cells [10]. DNA is a double helix structure consisting of two polymer strands that are made up of many nucleotide units. Each nucleotide is made up of 3 structures: a sugar, a phosphate group, and one of four nucleotide bases (cytosine, guanine, adenine, and thymine) which are typically abbreviated to the single characters C, G, A, and T respectively. The two strands are paired through hydrogen bonds based on their chemical structure. If cytidine appears on one strand, guanosine occupies the corresponding position on the other strand and similarly adenosine is paired with thymidine. DNA strands also have directionality; a single stranded DNA will have a 3'-end and a 5'-end [27]. These ends are dependent on which carbon atom in the sugar molecule the next phosphate in the chain is attached.

In this thesis, in addition to synthetic data, three types of biological data were used for classification: PCR primer DNA, Mitochondrial DNA, and Intron and Exon DNA. These data types are discussed in Chapter 3.

1.2 The Human Genome

The three billion letter code that distinguishes *Homo sapiens* from other species is known as the human genome. The first draft of the human genome was released in June 2000 from the Sanger Center in Cambridge, England and was considered a great step in the field of biology [10]. However, with this great step comes many questions that have yet to be answered by scientists, “Which genes are involved in preventing disease?” and “How can we slow down the aging process?” [10]. In order to answer these questions, much effort needs to be put into the analysis of biological data, as there is an enormous gap between the availability of sequence data and an understanding of these questions [10]. DNA sequence classification can answer these questions via the classification of features such as sequence patterns or motifs, substring statistics, GC-content, and various other features dependent on the problem at hand. These features are discussed in Chapter 2. Due to the immense search space of the human genome, evolutionary computation becomes a viable choice of classification techniques.

1.3 Evolutionary Computation

An evolutionary algorithm is a population based optimization algorithm; it utilizes mechanisms inspired by biological evolution such as reproduction, mutation, recombination and selection. Population evolution occurs after repeated application of these evolutionary mechanisms.

Algorithm 1 (Structure of an Evolutionary Algorithm [2])

Create an initial population.

Repeat

Select parents from the population with a fitness bias.

Copy the parents to make children.

Perform crossover on the children (optional).

Mutate the resulting children (probabilistic).

Place the children in the population.

Evaluate the fitness of the children.

Until Satisfied.

In the context of evolutionary computation, crossover and mutation are referred to as *variation operators*, they are analogous to biological crossover and biological mutation that occurs during reproduction.

There are several types of crossover that can be used to procure variability from one generation to the next. If an organism is represented by a string (called a gene), a crossover point is a randomly generated point in that string. Single-point crossover is performed by choosing a single crossover point on both parent strings. After copies of the parents are made, all data beyond the crossover point in both parents is swapped, producing two children. Single-point crossover leads to problems in that loci near one another are kept together with a higher probability than those that are farther apart [2]. This is a problem because single-point crossover is not disruptive enough to the gene to allow for sufficient exploration of the fitness landscape (discussed later in this section). Two-point crossover uses two crossover points and everything between those two crossover points is swapped in the resulting two children, see Figure 1.1. Multi-point crossover reduces the problem of loci near one another having a higher probability of being kept together.

Figure 1.1: Two-point crossover.

Uniform crossover is accomplished by putting a probability of α at each item's position in the gene to decide whether to take genetic information from the first parent or the second parent. This is a computationally expensive process as there are a great number of random numbers needed [2]. Adaptive crossover changes with the evolution of the population. All population members have an associated template string made up of 0's and 1's that determines which item in the parent's genes goes to which child. In positions where the template has a value of 0, items go from the first parent to the first child and the second parent to the second child; in positions where the template has a value of 1, items go from the first parent to the second child and from the second parent to the first child [2], see Figure 1.2. The template for Parent 1 is always chosen to do the crossover and the template associated with each population member is also evolved from one generation to the next using its own crossover and mutation operators [2]. Adaptive crossover carries the problem of being a two-time scale problem, that is, it generally takes less time to find the fit genes with the crossover templates than to find the crossover template in the first place [2].

Figure 1.2: Adaptive crossover.

Null crossover is equivalent to no crossover; the children are copies of the parents, it is used mainly when there are several crossover operators being used or to debug a program [2].

Similar to crossover, there are several types of mutation that can be used in evolutionary computation. Single-point mutation randomly selects a point on a string and randomly changes it to another value; similarly, multi-point mutation randomly selects a fixed number of points on a string and randomly changes them to another value. Point mutations make minimal changes to the gene but can be very powerful when used in evolving more complex structures. In the evolution of complex structures, some parts of a structure often end up being unused. When point mutation occurs in these unused parts, called neutral mutations, the structure can build up and eventually become reconnected to other points in the space with a single mutation. Probabilistic mutation performs a mutation with a fixed probability at each position. A Lamarckian mutation is performed by looking at all possible combinations of some fixed number of point mutations or less and using the one that results in the highest fitness value. Both probabilistic and Lamarckian mutation are computationally expensive. Similar to null crossover, null mutation is equivalent to no mutation.

A fitness landscape is the graph of the fitness function over the space of all data structures that could appear as members of the population in some evolutionary algorithm [2]. The fitness function quantifies the optimality of a solution and indicates which solutions are better than others [2]. The crossover operator allows for a broad search of the function space as it mixes and matches diverse genes, whereas the mutation operator allows for a local search of the function space as it makes small random changes to the genes. The combination of these two operators reduces the chance, with sufficient evolutionary runs, that the population will become trapped in local optima and will find the global optimum.

A local optimum is a solution that is optimal based on neighboring solutions in the fitness function and the global optimum is a solution that is optimal over the entire domain of the function. There are often many local optima in a fitness landscape, which is why population members often get caught in local optima if not enough diversity is introduced into the population. When more diversity is introduced, the population is given the opportunity to explore more areas in the fitness landscape and is more likely to climb to the global optimum. This is why mutation and crossover are very important operators to an evolutionary algorithm, otherwise the optimal solution obtained would have a larger chance of being caught in local optima.

Origins of evolutionary computation can be traced back to many independent publications that simulate the process of evolution using a computer program [9]. Box [5], Friedberg [12, 13], and Bremermann [6] published early works on this topic. In the 1960s many significant publications appeared. In 1962 Fogel introduced evolutionary programming [11] and in the same year Holland introduced the genetic algorithm [16]. In Germany, Rechenberg and Schwefel introduced evolution strategies in 1965 [31] and 1968 [32]. The field of evolu-

tionary computation remained relatively unknown to the scientific community for almost 30 years and not until 1991 was the term evolutionary computation finally coined [9]. Lack of available powerful computer resources was the main reason the field remained unknown for so long.

Evolutionary algorithms were used for the simulations performed in this thesis for the purpose of DNA pattern recognition. Evolutionary computation has been used for DNA pattern recognition in several publications by David Corne [24, 8, 25, 30]; in [24], an evolutionary algorithm was used to uncover significant patterns that indicate preferences for microsatellites to appear nearby and other patterns which seem to inhibit microsatellite production. Microsatellites are repetitive regions of DNA that account for a substantial proportion of eukaryotic genomes [24].

1.4 Side Effect Machines

A finite state machine consists of an input alphabet, a collection of states, a transition function, and an initial state [2]. The transition function encodes how the automaton moves from one state to another depending on the current state and input [2].

A side effect machine is a finite state machine that consists of a collection of states, transitions between those states, and *side effects* associated with each state. The *side effect* of the finite state machine is implemented as a state counter. The input alphabet in this case is a four character string of DNA {C,G,A,T}. Due to the four-character string nature of DNA, there are four transitions out of each state. Each time a character of the DNA string passes through a state, a counter on that state is incremented. When the side effect machine

has finished processing a DNA sequence, the vector of counts is normalized to obtain a unit vector that is indexed by the states of the side effect machine. These count vectors are used as numerical feature sets for a string of DNA. The side effect machine always begins in state zero with a λ -transition that is representative of the string entering the side effect machine. Counters are reset to zero when a new string is about to be run through the machine. An example of a side effect machine is shown in Figure 1.3.

Figure 1.3: Example of a side effect machine with 4 states and the count vectors given by running three example strings of DNA through the machine.

1.5 Evolving Side Effect Machines

In order to evolve a side effect machine using evolutionary computation, a simple way to represent the side effect machine had to be designed. A linear array was produced with

number of rows equal to the number of states in the side effect machine and four columns representative of the transitions for bases {C,G,A,T}. See Figure 1.4 .

	C	G	A	T
0:	2,	3,	2,	0
1:	0,	2,	3,	3
2:	2,	0,	3,	0
3:	0,	3,	3,	0

Figure 1.4: Side effect machine representation.

Figure 1.5: Side effect machine diagram of representation shown in Figure 1.4.

Evolutionary mechanisms such as crossover and mutation are defined for this type of representation by treating each row of the table as a *character* for the string-based variation operators described in Section 1.3.

Two-point crossover was performed by swapping all rows between two *crossover points*

in the linear array. Two-point crossover is demonstrated in Figure 1.6. Two crossover points are randomly selected to be between states 2 and 3 and states 5 and 6 on the two parent machines, resulting in two offspring machines.

	C	G	A	T		C	G	A	T
0:	0,	7,	5,	2	0:	1,	3,	7,	7
1:	6,	7,	0,	1	1:	3,	4,	4,	0
2:	6,	7,	4,	3	2:	4,	1,	5,	4
3:	6,	0,	5,	2	3:	4,	5,	6,	1
4:	2,	1,	7,	5	4:	1,	6,	4,	0
5:	2,	0,	0,	7	5:	2,	0,	3,	1
6:	7,	0,	0,	5	6:	5,	5,	3,	5
7:	5,	0,	0,	1	7:	1,	1,	1,	0
(a) Two Parent Machines									
	C	G	A	T		C	G	A	T
0:	0,	7,	5,	2	0:	1,	3,	7,	7
1:	6,	7,	0,	1	1:	3,	4,	4,	0
2:	6,	7,	4,	3	2:	4,	1,	5,	4
3:	4,	5,	6,	1	3:	6,	0,	5,	2
4:	1,	6,	4,	0	4:	2,	1,	7,	5
5:	2,	0,	3,	1	5:	2,	0,	0,	7
6:	7,	0,	0,	5	6:	5,	5,	3,	5
7:	5,	0,	0,	1	7:	1,	1,	1,	0
(b) Two Offspring Machines									

Figure 1.6: Example of two-point crossover.

Mutation was done by randomly selecting and changing the transition numbers in the body of the linear array. One-point mutation changes one of the transitions, two-point mutation changes two of the transitions, and so on. If we randomly change the G transition leaving state 0 and the T transition leaving state 3 from Figure 1.4, a two-point mutation will have been performed. See Figure 1.7.

	C	G	A	T
0:	2,	①,	2,	0
1:	0,	2,	3,	3
2:	2,	0,	3,	0
3:	0,	3,	3,	②

Figure 1.7: Two-point mutation on side effect machine from Figure 1.4.

As described in Section 1.3, crossover and mutation allow for efficient exploration of the fitness landscape. A more detailed description of evolving side effect machines and an explanation of the fitness function used will appear in Chapter 3.

Chapter 2

Definitions and Theorems on Side Effect Machines

Feature selection is a technique, used in machine learning, of selecting a subset of pertinent features for assembling a robust population of individuals. Feature induction is a learning method that uses positive and negative examples of training data to *teach* a population to classify these examples by generating effective features. This chapter gives theorems and proofs of several valuable features that allow a robust set of side effect machines to evolve.

2.1 Tag Detector Side Effect Machine

Detection of a specific substring of DNA has many applications in medical research, from detecting specific genes in humans to recognizing characteristic genes in disease causing species. The tag detector side effect machine described below allows the identification of specific substrings of DNA and is an important feature in evolving side effect machines.

Theorem 1 (Tag Detector Side Effect Machine) *There is a side effect machine that can count the number of times a particular substring or set of substrings occurs in a DNA sequence.*

Proof 1 (Tag Detector Side Effect Machine)

For a substring of length n ,

1. Create a side effect machine with n states numbered consecutively in addition to the initial state-zero, resulting in a total of $n + 1$ states.
2. Draw transitions between each state representative of each character in the substring in the order they appear in the substring.
3. If the substring contains the prefix of the substring, transition back to a state where that prefix has already been seen.
4. When using the tag detector side effect machine to find more than one substring within a string, use any overlap of the substrings to transition from one substring to another.
5. Draw prefix transitions from each state after state-one to state-one unless the substring begins with repeating characters, in which case, draw a self-transition on the last state of the repeating characters.
6. *-transitions represent seeing a character other than the transitions already given. Draw these transitions from each state to state-zero, including a self-transition on state-zero, unless all four characters of the DNA alphabet have transitions out of that state.

Example 1 (Tag Detector Side Effect Machine) (See Figure 2.1)

Figure 2.1: Side effect machine used to detect the substrings G,G,T and G,A,T.

2.2 GC-content Side Effect Machine

GC-content (or guanine-cytosine content) is the percent of nitrogenous bases in a DNA sequence that are either guanine or cytosine. Guanine and cytosine have three hydrogen bonds and adenine and thymine have two hydrogen bonds connecting them. This makes the bond between guanine and cytosine more stable than that of adenine and thymine. Thus, a high GC-content string of DNA is more stable and has a higher melting temperature than

DNA with a low GC-content. This property is important in PCR primer experiments as the melting temperature of the pair of primers must be roughly the same for the reaction to be successful.

GC-content is a standard classification feature for biological sequences and the following theorem and proof demonstrates that side effect machines can use this feature if it is needed.

Theorem 2 (GC-Content) *There is a side effect machine that can calculate the percent GC-content of a particular DNA string.*

Proof 2 (GC-Content) We construct a two-state side effect machine where one state counts G and C content and the other state counts the A and T content of the DNA string.

Example 2 (GC-Content) (See Figure 2.2)

Figure 2.2: Side effect machine used to calculate GC-content of a DNA string.

2.3 Substring Statistic Side Effect Machine

Substring statistics have several applications in fields such as data compression and pattern recognition. In the context of this thesis, given a DNA string, the substring statistics problem consists of calculating the number of times each and every distinct substring or pattern of a given length occurs in that string (with overlap). Many nucleotide patterns have

biological significance such as encoding the secondary structure of a protein or providing labels for protein delivery to specific areas in the cell. The following describes the methods implemented to calculate substring statistics of any length.

Definition 1 (Graph) *A graph consists of a set $V(G)$ of vertices, a set $E(G)$ of edges, and a mapping associating each edge to an unordered pair of vertices [35].*

Definition 2 (Walk) *A walk in a graph G of length k consists of an alternating sequence of vertices $x_i \in V(G)$ (not necessarily distinct) and edges $e_i \in E(G)$ such that the ends of e_i are exactly x_{i-1} and x_i , where $i = 1, 2, \dots, k$ [35].*

Definition 3 (Path) *If the edges $e_1, \dots, e_k$ in a walk are distinct, then it is referred to as a path from x_0 to x_k [35].*

Definition 4 (Connected Graph) *If a path exists for every pair of vertices in G , then G is called a connected graph [35].*

Definition 5 (Line Graph) *A line graph $L(G)$ of a graph G is a graph that represents the adjacencies between the edges of G [35].*

Example 3 (Line Graph) (See Figure 2.3)

Figure 2.3: Example of a line graph construction.

Definition 6 (De Bruijn Sequence) *A cyclic sequence over a given alphabet with size k for which every possible subsequence of length n appears as a sequence of consecutive characters exactly once [35]. Each De Bruijn sequence has length k^n .*

Definition 7 (Eulerian Cycle) *A closed path through a graph which visits every edge once [35].*

Theorem 3 (Eularian) *A finite graph with no isolated vertices is Eularian if and only if it is connected and every vertex has in-degree equal to the out-degree [35].*

Proof 3 (Eularian) See [35].

Binary De Bruijn graphs can be constructed using line graphs where $|A| = k$ is the number of characters in the alphabet (for binary $A = \{0, 1\}$, so $|A| = k = 2$) and n is the length of every possible subsequence that appears as a sequence of consecutive characters exactly once. The dimension of the line graph is $n - 1$. Thus, each edge in the $(n - 1)$ -dimensional De Bruijn graph corresponds to a sequence of n -digits. See Figure 2.4.

Figure 2.4: Line graph construction of the three smallest binary De Bruijn graphs.

The number of vertices is defined by $V = k^{n-1}$ and edges are defined by $E = (s, t)$ where the length- $(n-2)$ suffix of s is the prefix of t . Each De Bruijn sequence has length k^n .

In Figure 2.4, the first (1-dimensional) line graph with $k = 2$ and $n = 2$ has 2^1 vertices and the length-zero suffix of s is the prefix of t for all edges (no overlap). There are 2^2 ($n = 2$)-digit sequences that appear exactly once, $\{00, 01, 11, 10\}$. In the second (2-dimensional) line graph with $k = 2$ and $n = 3$, there are 2^2 vertices and the length-one suffix of s is the prefix of t for all edges. There are 2^3 ($n = 3$)-digit sequences that appear exactly once, $\{000, 001, 011, 111, 110, 101, 010, 100\}$. For the third (3-dimensional) line graph

in Figure 2.4 with $k = 2$ and $n = 4$, there are 2^3 vertices and the length-two suffix of s is the prefix of t for all edges. There are 2^4 ($n = 4$)-digit sequences that appear exactly once, $\{0000, 0001, 0011, 0111, 1111, 1110, 1100, 1001, 0010, 0101, 1011, 0110, 1101, 1010, 0100, 1000\}$.

Each of the three graphs in Figure 2.4 has a Eulerian cycle because every vertex has an in-degree that is equal to the out-degree.

A rotating drum is shown in Figure 2.5, it represents the 3-dimensional binary line graph from Figure 2.4. It is read circularly and illustrates the Eulerian path through all transitions and vertices of the 3-dimensional binary line graph. Notice that each three-character substring appears exactly twice whereas each sixteen four-character ($n = 4$) substring appears exactly once. Note that the De Bruijn sequence has length 2^4 .

Figure 2.5: Rotating drum diagram of the 3-D binary line graph shown in Figure 2.4.

Theorem 4 (Substring Statistics) *There is a side effect machine that accurately counts the number of each type of substring of length n , for any n .*

Proof 4 (Substring Statistics) The construction of a De Bruijn sequence automaton that can be transformed into a side effect machine can be used to count small substring statistics. A De Bruijn sequence automaton refers to a De Bruijn graph that is a self-driving machine in the same way that a side effect machine is DNA-driven with an input string. This transformation is done using an entry tree that allows a substring of size n to enter the De Bruijn automaton.

An example with a substring of size $n = 2$ of an entry tree, Eulerian cycle, and a De Bruijn rotating drum diagram of a path through the Eulerian cycle is shown in Figure 2.6. The DNA alphabet has a size of $k = 4$ as $A = \{C, G, A, T\}$. There are 4^1 vertices and there are 4^2 ($n = 2$)-digit sequences that appear exactly once (shown in Figure 2.6).

Larger DNA substring sizes are not shown as the $n = 3$ substring results in a De Bruijn sequence with length 4^3 .

Figure 2.6: a) Entry tree for substring of size $n = 2$, b) Eulerian cycle for substring of size $n = 2$, c) De Bruijn rotating drum diagram for substring of size $n = 2$ demonstrating the cycle $CG \rightarrow GC \rightarrow CA \rightarrow AC \rightarrow CT \rightarrow TT \rightarrow TA \rightarrow AT \rightarrow TG \rightarrow GG \rightarrow GA \rightarrow AA \rightarrow AG \rightarrow GT \rightarrow TC \rightarrow CC \rightarrow \text{START}$.

The methods described in this chapter show that side effect machines are an ideal structure for detecting common features in DNA strings that are of biological importance.

Chapter 3

Experimental Design

3.1 Fitness Function

A fitness function is used to measure the quality of solutions or quantify which solutions are better than others. It is used to drive evolution of a population of individuals. In this study, side effect machines require a fitness function in order to be trained with an evolutionary algorithm. An obvious approach in choosing a fitness function for evolving side effect machines is the use of supervised learning. Supervised learning is a machine learning technique that allows a fitness function to be learned from training data. Training data is composed of input with desired or correct output. In the context of this study of DNA classification, the training data consists of DNA sequences that have a known classification based on functionality, origin, or region.

As described in Chapter 1, side effect machines map DNA strings to state count vectors. These count vectors are used as a numerical feature set for all DNA strings that are run through a side effect machine. The count vectors are clustered using k -means clustering,

described later in this section, and the k -means clustering of the count vectors is compared to the known, correct clusters in the training data. The fitness of a side effect machine is computed with the Rand index, comparing the correct classification with that obtained by k -means clustering the feature vectors produced by the side effect machine. The Rand index is also described later in this section.

Figure 3.1: Flowchart demonstrating the side effect machine fitness function.

3.1.1 k -Means Clustering

k -means clustering is an unsupervised learning technique used to cluster objects based on their characteristics into k partitions. If the algorithm works well, then the data items within a cluster are similar to each other and not similar to items in the other clusters. The

algorithm is used to find the centers of natural clusters that occur in data sets. The objective of the algorithm is to find plausible clusters in the data. Side effect machines are trained to create count vectors that are particularly easy for the k -means method to correctly cluster.

Algorithm 2 (k -means Clustering—See Figure 3.2):

1. *Randomly choose k distinct points from the data set as initial cluster centers (indicated by black circles).*
2. *Assign each point to the cluster center whose center it is closest to, breaking ties at random. If a point falls exactly between two cluster centers, it is randomly assigned to a cluster—this is a very rare occurrence.*
3. *Re-compute cluster centers as the average of all points in the cluster (indicated by black squares).*
4. *Repeat steps 2-3 until no points change their cluster assignment or B cycles have occurred. B is a number chosen by the user; indefinite cycles are extremely rare and B guards against them.*

Figure 3.2: k -means clustering algorithm.

3.1.2 Rand Index

The Rand index is used to compare two partitions of a data set, it is a measure of similarity between those two partitions.

Definition 8 (Rand index) Let D be a set of data and let P and Q be two partitions of this data into classes. The Rand index comparing P and Q is the fraction of pairs from D that are either in the same class in both P and Q or are in distinct classes in both P and Q (the fraction of correct pairs).

Example 4 (See Figure 3.3) For this example, D is the set of $n = 6$ data points. Let P be the result from k -means clustering and let Q be the known correct classification of the data. There are a total of $\binom{6}{2} = 15$ possible pairs in D : (1,2); (1,3); (1,4); (1,5); (1,6); (2,3); (2,4); (2,5); (2,6); (3,4); (3,5); (3,6); (4,5); (4,6); (5,6). If a is defined as the number of pairs

in D that are in the same class in P and the same class in Q , then $a = (1,2); (5,6) = 2$ pairs.

If b is defined as the number of pairs in D that are in different classes in P and different classes Q , then $b = (1,3); (1,4); (1,5); (1,6); (2,3); (2,4); (2,5); (2,6); (3,5); (3,6) = 10$ pairs.

Thus the Rand index is 0.8.

$$\text{Rand index} = \frac{a + b}{\binom{n}{2}} = \frac{2 + 10}{\binom{6}{2}} = 0.8$$

Figure 3.3: Rand index example.

The Rand index is a measure of similarity of two partitions of the same data set. When the Rand index has a value of 1.0 the two classifications are in complete agreement.

There remains a problem with the value generated from the Rand index formula. The Rand index is not useful statistically when there are more than two classes because its value approaches 1.0 as the number of classes in the partitions are increased. The adjusted Rand index provides a better statistical measure, however, for the purpose of this thesis (using the Rand index as a fitness score), the simpler Rand index formula is sufficient in that it correctly orders the comparisons of the partitions. A fitness score only requires an ordering of the fitness values. The adjusted Rand index is described in the following example [37].

Example 5 (See Figure 3.3) Continued from the previous example where D is the set of $n = 6$ data points with two partitions P and Q , shown in Figure 3.3. There are a total of $\binom{6}{2} = 15$ possible pairs in D , $a = 2$ and $b = 10$, defined in the previous example. Two new variables are introduced: c is defined as the number of pairs in the same class in P and different classes in Q and d is defined as the number of pairs in different classes in P and the same class in Q . Then $c = (3,4) = 1$ pair and $d = (4,5); (4,6) = 2$ pairs. Thus using the formula below, the adjusted Rand index is calculated as 0.44, which can be interpreted as the k -means result and the known classification of the data are in 44% agreement.

$$\text{Adjusted Rand index} = \frac{a - \frac{(a+c)(a+d)}{\binom{n}{2}}}{\frac{(a+c)+(a+d)}{2} - \frac{(a+c)(a+d)}{\binom{n}{2}}} = \frac{2 - \frac{(2+1)(2+2)}{\binom{6}{2}}}{\frac{(2+1)+(2+2)}{2} - \frac{(2+1)(2+2)}{\binom{6}{2}}} = 0.44$$

3.2 Synthetic Data

Before attempting to classify biological data, parameter studies were done using well characterized synthetic data. Biological data is often noisy and makes the implementation of a new technique like SEMs difficult to study without prior controlled data application.

3.2.1 Entropy Bounded Data

Entropy bounded data was created using a self-driving Markov automaton. See Figure 3.4 for an example.

State	P(C)	P(G)	P(A)	P(T)
0	0.439	0.023	0.374	0.164
1	0.255	0.014	0.284	0.447
2	0.275	0.067	0.101	0.556
3	0.176	0.536	0.242	0.046

example output:
TACTCTTAAGCGAATTAACAAGACC-
GAACAGACTTCCGTTGTTTGACTGT-
ACAGTGTGCAGCCCCCAGTGCAGCG-
AACGAATTTCAACAGGTGAGATTTT

Figure 3.4: Example of a self-driving Markov automaton used to create the first category of data in the fifth data set with a Shannon entropy of 1.8 and an example DNA string output. Note that each state has an associated distribution on $\{C,G,A,T\}$ used to emit symbols.

A self-driving Markov automaton is a stochastic finite state transducer that is used to sample a probability distribution on a set of DNA strings. A base is generated according to a local probability distribution in the current state and it is used to drive transitions to

the next state. In other words, given the present state, future states are independent of the past states. The machine begins in state-zero and emits a symbol using the distribution at state zero. This symbol is then used to drive a state transition to a new state where a new symbol is generated. If there was a uniform probability distribution on each state, then the generated data would be indistinguishable from data generated from a uniform random DNA base-generator.

Control of the entropy of the distributions associated with each state permits an adjustable degree of randomness in the synthetic data. The Shannon entropy equation, used to limit the randomness of the local distributions is:

$$E = - \sum_{i=1}^m p_i \log_2(p_i).$$

For this particular study, in which the entropy for DNA bases is being calculated, $m=4$ due to the four character alphabet of DNA or $p_1 = P(C)$, $p_2 = P(G)$, $p_3 = P(A)$, and $p_4 = P(T)$. The Shannon entropy is a standard measure of randomness of a distribution. It quantifies the information contained in a message in bits per symbol on average. The value of the entropy of a distribution is the number of bits required to describe a sample from the distribution on average. A uniform, random probability distribution of a DNA sample can be described with 2 bits (00,01,10,11) by the calculation below.

$$P(C) = P(G) = P(A) = P(T) = 0.25$$

$$E = -4(0.25)\log_2(0.25) = 2$$

For each of the five data sets, three sets of 60 sequences were generated using three different self-driving Markov automaton for each data set. Each sequence has a length of

100 bases. The entropy of the probability distributions at each state for the five data sets was set to be at most 1.0, 1.2, 1.4, 1.6, and 1.8 for data sets 1, 2, 3, 4, and 5 respectively. These probability distributions were chosen by generating four integers in the range 1-100 and dividing by the sum of the integers to get probabilities for each of the four DNA bases. In order to meet the maximum entropy boundary conditions for the five data sets, probability distributions were generated until the entropy bound was satisfied.

Experimental Design

As outlined in Chapter 1, side effect machines are represented by a linear array of states and evolution was performed using two variation operators: two point crossover of the list of states and point mutation of one or more state transitions. For each of the five synthetic data sets, 100 evolutionary runs were completed using size seven single tournament selection, and evolution was run for a maximum of 10,000 mating events (100 generations). Size seven single tournament selection is a method of genetic selection where seven individuals (side effect machines) are chosen at random from the population and the two individuals with the best fitness breed (parents replicate and crossover) and replace the two individuals with the worst fitness. The parameters: number of states, population size, and mutation rates were explored to characterize their effect on performance. Thus a factorial experiment was completed using each of the five synthetic data sets with population sizes 10, 100, and 400 side effect machines; 4-, 12-, and 20-states; and 1-, 2-, and 3-point mutations for a total of $3^3 \times 5 = 135$ combinations of factors. After results were obtained on the initial experiments, an additional parameter study was completed with 28-states, 1-, 2-, and 3-point mutations, and a population size of 10.

Algorithm 3 (Evolutionary Algorithm for Entropy Bounded Data)

1. *Randomly generate an initial population of side effect machines with population size 10, 100, or 400 and with 4-, 12-, or 20-states.*
2. *For each of the five synthetic data sets, run the 180 DNA sequences (consisting of three groups of 60 generated from three distinct Markov automaton) through each side effect machine in the population.*
3. *Feature vectors are generated and saved for each of the 180 DNA sequences run through each side effect machine in the population.*
4. *For each side effect machine in the population, permit k-means clustering of the 180 feature vectors (where $k = 3$).*
5. *Compare the k-means clustering result with the known correct classification of the DNA sequences, returning the Rand index of the comparison as the fitness. This is performed for each side effect machine in the population.*
6. *Perform size seven single tournament selection (described earlier in this section) on the entire population of side effect machines.*
7. *Perform 1-, 2-, or 3-point mutation, selected with equal probability, on the two resulting children that replace the two individuals in the tournament with the worst fitness and place the children into the population.*
8. *Evaluate the fitness of the two children.*

9. Repeat Step 6–8 for a maximum of 10,000 mating events (less if fitness of 1.0 is reached before 10,000 mating events).

10. Repeat Steps 1–9 to complete 100 evolutionary runs.

3.2.2 GC-Content Data

See Section 2.2 for description of GC-content.

Experimental Design

DNA sequences with given percentages of GC-content were created for experimentation. Four sequences, 200 nucleotides in length, were created with a particular G/C to A/T percentage, these sequences were then shuffled and a cutoff point was chosen uniformly at random resulting in sequences between 100 and 200 nucleotides in length. As a consequence of choosing this cutoff point, all of the following GC-content percentages are approximate. Two experiments were created with two classes of sequences for each. For the first experiment, 500 sequences with GC-content of 55% were put into one class and 500 sequences of GC-content of 45% were put into the other class. For the second experiment, 500 sequences of GC-content of 60% were put into one class and 500 sequences of GC-content of 40% were put into the second class. A population of 10 side effect machines and a maximum of 10,000 mating events was used for both data sets, these parameters were chosen based on the results of the synthetic data experiments. A 2×2 factorial experiment was completed with side effect machines with 4-, 12-, and 20-states and 1- and 2-point mutations.

Cross-validation was performed on the GC-content data to detect whether the side effect machines *generalized* or *memorized* the DNA sequences. The desired result is the generalization of the data, in other words, finding the patterns in the DNA string and separating the data into appropriate classes. Memorizing the data is an undesirable result as it implies the side effect machines simply memorized the training data and did not *learn* how to classify the data based on their particular characteristics. Cross-validation is implemented by reserving *test* data (that is not used to train the machine) to later validate a model generated from the *training* data [2]. For the GC-content experiment, four sets of *test* sequences with the same G/C to A/T percentages were generated in the same fashion as the *training* data for cross-validation.

3.2.3 Stochastic Fitness Experiment with Motif Location Data

A stochastic fitness algorithm was used in an attempt to reduce the likelihood of the side effect machines memorizing the data sets instead of generalizing them. This was done by randomly selecting a subset of the training sequences without regard for the number of sequences taken from each class. When the initial population is chosen, solutions are created then mutated and selected for those with higher fitness until an acceptable solution has been found. If the majority of the evolutionary runs of these distinct initial populations result in the same solution then that solution is most likely representative of the global optima of the fitness function. In other words, by selecting a fraction of the DNA sequences for each evolutionary run, a different fitness landscape is produced in each generation resulting in a greater likelihood of persistent optima representing the global

optima of the fitness landscape. This *partial* subset of the data is more likely to cross-validate with a higher fitness than the *full* data set because memorization of the data becomes harder as a different initial population is present for each new evolutionary run.

Experimental Design

Two classes of 500 sequences of DNA were generated uniformly at random with a maximum length of 250 bases. The first class contains the 10 character substring *CCGATATGGG* and the second class does not contain that substring, the result, a pool of 1000 sequences to choose from. For this experiment, the *full* data set of 1000 sequences were classified and compared to the *partial* data set of 200 different sequences that were randomly chosen for classification in each of the 100 evolutionary runs. The *full* and *partial* data sets were then cross-validated with test data that was created in the same manner as the training data. A population of size 10 and a maximum number of mating events was set at 20,000 for every run; 12-, 20-, and 28-states were tested with 1- and 2-point mutations.

3.3 Biological Data

After application of side effect machines to synthetic data, this technique was used to classify biological data sets. Side effect machines were trained with an evolutionary algorithm to classify PCR primer DNA, mitochondrial DNA and intron and exon DNA.

3.3.1 PCR Primer DNA

The polymerase chain reaction (PCR) is an in vitro technique that allows the amplification of a specific region of DNA that lies between two known regions of a DNA sequence [27].

PCR amplification is achieved with the use of oligonucleotide primers. Oligonucleotide primers are short single-stranded DNA molecules that are complementary to the ends of a sequence of DNA [27]. A double helix strand of DNA is heated which causes the hydrogen bonds between the nitrogenous bases to be disrupted, resulting in two single strands of DNA that act as templates. Since smaller segments of DNA anneal faster than longer segments [3], when the DNA is subsequently cooled, the short stranded primers bind to the single strands of DNA. Under suitable reaction conditions and the presence of deoxynucleoside triphosphates, DNA polymerase, extends the primer to construct a complementary DNA strand to the template in the 5'-end to 3'-end direction [27]. This results in two double-stranded DNA molecules. This process is repeated until the exhaustion of reagents and enzyme [27].

The polymerase chain reaction has made tremendous impacts in the fields of medicine and forensics. The ability to use small samples of DNA for determining a larger sequence of DNA enables prompt diagnosis of many infectious diseases, determination of genetic defects in fetuses or neonates, effective monitoring of cancer chemotherapy, improved matching of donor organs to recipients, and solving paternity and criminal cases [17]. In diagnosing infectious diseases, PCR allows for rapid detection of a small number of microorganisms in a patient specimen by using primers that are complementary to a specific characteristic gene of a particular disease causing species [17]. Earlier methods of

diagnosing bacterial disease required either the lengthy process of growing microbes followed by histologic and chemical analyses (which was ineffective if the patient needed immediate treatment) or a test for antibodies produced by the patient (which was ineffective if the patient had not yet produced antibodies) [17]. In cancer treatment, PCR is used to find recurrences or metastases when a patient is undergoing chemotherapy by locating DNA rearrangements that are present in cancer cells but are not found in normal, cancer-free, cells [17]. In paternity tests, PCR is used to amplify multiple genes; in forensics, it is used to amplify small DNA specimens found at crime scenes [17].

Figure 3.5: One complete cycle of the polymerase chain reaction [17]. Starting with unamplified DNA and resulting in the doubling of the target region of DNA.

Experimental Design

The PCR primers used in these experiments were designed for a genetic mapping project for *Zea mays* (corn) using a software program called Primer 3 from the NCSA Biology

Workbench. Many software programs like Primer 3 that pick primers to amplify known DNA sequences are available for free download. Side effect machines are applied to the problem of classifying primers that were designed by Primer 3 to have good biophysical properties; some which amplify correctly and others that do not amplify correctly. There are a few obstacles in finding good primers. Since primers work in pairs, a good primer may end up being scored as a bad primer because of flaws in its partner. Also, if a primer anneals in more than one place within the target genome, DNA other than the target may be amplified. This problem is exhibited highly in the corn genome as it possesses a large number of *transposons* (far more than the average organism). *Transposons* are a type of genetic parasite - a stretch of DNA that makes copies of itself in the genome. Since the entire genome of an organism is present when the PCR reaction begins, the primer sequence that is to amplify a particular target may, because of sequence duplication, confound the PCR reaction. In the genetic mapping project for *Zea mays*, machine learning can be used to detect and defeat this confounding sequence redundancy.

The PCR primers for this experiment were selected from the thousands of primers that were designed and tested from the *Zea mays* mapping project. They consisted of two classes of 100 sequences, 100 sequences that were classified as *good* at amplification and 100 sequences were classified as *bad* at amplification. 100 evolutionary runs were performed using size seven single tournament selection, and evolution was run for a maximum of 20,000 mating events with a population of size 10. A parameter study was done to find the optimal number of states and mutations that achieve the best fitness; 4-, 8-, 12-, 16-, 20-, 24-, 28-, 32-, 36-, and 40-states and 1- and 2-point mutations were tested, resulting in 20

experiments. A second set of 200 primers (half *good*, half *bad*) were selected for cross-validation.

3.3.2 Mitochondrial DNA

Mitochondrial DNA (mtDNA) is found in organelles called mitochondria. The coding system for mtDNA is similar but not exactly the same as that for nuclear DNA [19].

Unlike nuclear DNA, which is inherited through both parents, mitochondrial DNA is only inherited from the mother, resulting in clonal inheritance that can be very useful in identity testing [19]. One mitochondrion contains up to 10 copies of mtDNA, there are between 300 and 400 mitochondria in every cell, giving ten million billion mitochondria in the human body [20]. This abundance also makes mtDNA more suited to identify older skeletal remains than nuclear DNA because it is often easier to collect.

Experimental Design

The mitochondrial data set consisted of six sets of 30 sequences that were taken from six distinct organisms. These organisms are *Melipona bicolor* (stingless bee) [33], *Chirocentrus dorab* (wolf-herring) [18], *Salanx ariakensis* (ice fish) [18], *Caulophryne pelagica* (fanfin anglerfish) [26], *Trachypithecus obscurus* (dusky leaf monkey) [22], and *Lugensa brevirostris* (Kerguelen petrel) [34].

100 evolutionary runs were performed using size seven single tournament selection, and evolution was run for a maximum of 20,000 mating events with a population of size 10. A parameter study was done to find the optimal number of states and mutations that achieve the best fitness; 4-, 8-, 12-, 16-, 20-, 24-, 28-, 32-, 36-, and 40-states and 1- and 2-point

mutations were tested, resulting in 20 experiments. Cross-validation was not applied to the mitochondrial data as test data was not available and the goal of this experiment was to see if side effect machines could learn how to classify mitochondrial DNA.

3.3.3 Intron and Exon DNA

During protein synthesis, regions of DNA are copied into an intermediate molecule called ribonucleic acid (RNA), this process is commonly known as transcription. RNA is similar to DNA except for a few structural differences: RNA is usually single stranded, contains ribose sugar, and contains the nucleotide base uracil instead of thymine. In the process of transcription, a pre-initiation protein complex binds to the DNA strand around the promoter region which is just upstream of the start codon (a three letter code that signals the beginning of the translation process) of the gene that is to be expressed [10]. A gene is a sequence of DNA, approximately 100-5000 nucleotides in length, that codes for the manufacture of a protein molecule [10]. The pre-initiation protein complex attracts a protein, RNA polymerase, which causes the two strands of DNA to disassemble from the start codon to the stop codon (three letter code that signals the end of the translation process). The RNA polymerase then binds to the DNA and moves along the gene from the 5'-end to the 3'-end producing a strand of messenger RNA (mRNA) that is left free-floating in the cell nucleus [10]. The messenger RNA is exported out of the nucleus and into the cell cytoplasm where it encounters a ribosome that translates every nucleotide triplet in the RNA sequence into one of 20 known amino acids [17]. This chain of amino acids forms a protein molecule. Before RNA is completely processed into mature mRNA, it

is termed precursor mRNA (pre-mRNA) [17]. Pre-mRNA includes two different kinds of segments: introns and exons. In the process of pre-mRNA changing into mature mRNA, introns are removed through splicing and exons remain in the final mRNA product [19]. Introns are the non-coding sections of pre-mRNA and enable alternative splicing of a gene [1]. Alternative splicing includes introns being retained in the mRNA, certain exons being spliced out of the pre-mRNA, and the use of alternative 5' and 3' sites for an exon which results in either a frame shift or insertion/deletion of amino acids [1]. Alternative splicing is useful in that it allows for several different proteins that share part of their sequences in common to be translated from a single gene, in other words, it allows for genetic information to be stored more efficiently.

Figure 3.6: DNA to protein formation through splicing and excision of introns out of pre-mRNA [19].

Experimental Design

Intron and exon DNA data sets consisted of two classes of 100 sequences, 100 sequences were made up of introns and 100 sequences were made up of exons. The data sets were taken from a subset of the known genes in *Zea Mays* (corn). The DNA for the introns and exons was identified by comparison of genomic DNA with captured RNA.

100 evolutionary runs were performed using size seven single tournament selection, and evolution was run for a maximum of 20,000 mating events with a population of size 10. A

parameter exploration was done to find the optimal number of states and mutations that achieve the best fitness; 4-, 8-, 12-, 16-, 20-, 24-, 28-, 32-, 36-, and 40-states and 1- and 2-point mutations were tested, resulting in 20 experiments. Similar to the mitochondrial data, the intron and exon DNA data sets did not undergo cross-validation due to a lack of available test sequences. The objective was to find which parameter values allowed for rapid progress on this biological data rather than to solve the classification problem. There is a strong contrast in good parameter choices between biological and non-biological data.

Chapter 4

Results and Discussion

4.1 Results and Discussion for Synthetic Data

Entropy bounded data sets, GC-content data sets, and the stochastic fitness experiment with motif location data results are stated and explained in the following three sections.

4.1.1 Entropy Bounded Experiments

For data sets 1 and 2 with maximum state entropies of 1.0 and 1.2 the best fitness was 1.0 representing perfect classification in all replicates. This suggests that the first two data sets were very easy for side effect machines to classify but also suggests that side effect machines may be an effective representation for the DNA classification problem. If the side effect machines had not passed this simple test there would have been no reason to continue this study. Due to the trivial nature of data sets 1 and 2, analysis and figures for these data sets have been omitted. In an applied problem, perfect classification would be ideal but when classifying synthetic data, it is uninteresting. For data sets 3 and 4 with

maximum entropies of 1.4 and 1.6, some parameter settings were able to obtain perfect fitness scores of 1.0. For data set 5 with maximum entropy of 1.8 the highest fitness achieved was just over 0.95. Initial parameter testing results are shown in Figures 4.1, 4.2, and 4.3. This study was intended to compare different parameter settings and the basis for comparison in each plot was chosen to maximize contrast.

Figure 4.1: Parameter study results for data set 3.

Probability best classifier has accuracy 0.98, data set 4

Figure 4.2: Parameter study results for data set 4.

Figure 4.3: Parameter study results for data set 5.

Figures 4.1, 4.2, and 4.3 show a 95% confidence interval on the probability that a run will achieve a given fitness using the Gaussian approximation to the binomial distribution applied to the 100 trials performed for each set of parameters. The threshold fitness for comparison is chosen to be the largest, to two decimals, that permits the worst parameter setting to achieve that fitness at least once. The thresholds are 1.0 for data set 3, 0.98 for data set 4 and 0.86 for data set 5. These figures are sorted in descending order of estimated probability of achieving the threshold fitness. In general, for all three of the higher entropy data sets, the choice of parameter settings made a significant difference in performance. The best population size parameter setting was found to be 10 in the parameter study

performed. This suggests that the evolutionary algorithm for side effect machines exhibits small population effects described in [4]. In [4], very small population sizes (4-7) are compared with population sizes of up to 1000 for several genetic programming problems: 4-parity using parse trees, Tartarus using ISAc lists, and several versions of plus-one-recall-store (PORS) using parse trees. The substantial changes in evolution found in [4] caused by small population sizes is called *small population effects*.

After results from the initial parameter search were obtained, a second experiment was done with data sets 3, 4, and 5 using only a population size of 10; 12-, 20-, and 28-states and 1-, 2-, and 3-point mutations. In Figures 4.4, 4.5, and 4.6, 95% confidence intervals and best-of-run fitness are shown for data sets 3, 4, and 5. For all three data sets, the best runs were clearly achieved with twelve states with the best number of mutations varying. It can be noticed that the number of mutations with the best performance in Figures 4.4, 4.5, and 4.6 and Figures 4.1, 4.2, and 4.3 are slightly different, this is because their relationship is non-linear and thus the parameter tradeoff is different for these two different measures.

Entropy 1.4 synthetic data, parameter study

Figure 4.4: 95% confidence intervals and best-of-run fitness for data set 3.

Figure 4.5: 95% confidence intervals and best-of-run fitness for data set 4.

Entropy 1.8 synthetic data, parameter study

Figure 4.6: 95% confidence intervals and best-of-run fitness for data set 5.

Figure 4.7: Fitness scores by generation for data set 3, 12-states.

Figure 4.8: Fitness scores by generation for data set 3, 20-states.

Figure 4.9: Fitness scores by generation for data set 3, 28-states.

Figure 4.10: Fitness scores by generation for data set 4, 12-states.

Figure 4.11: Fitness scores by generation for data set 4, 20-states.

Figure 4.12: Fitness scores by generation for data set 4, 28-states.

Figure 4.13: Fitness scores by generation for data set 5, 12-states.

Figure 4.14: Fitness scores by generation for data set 5, 20-states.

Figure 4.15: Fitness scores by generation for data set 5, 28-states.

4.1.2 GC-Content Experiments

The first experiment was to classify sequences with a GC-content of 55% from sequences with a GC-content of 45% and the second experiment was to classify sequences with a GC-content of 60% from sequences with a GC-content of 40%. These two experiments had very similar results with the exception of the second experiment being much easier for the side effect machines to solve. Due to this particular result, figures for the second experiment on GC-content have been omitted.

Once again, the side effect machines were able to classify the GC-content with a high level

of accuracy. On average the best runs were achieved with four states and the number of mutations as with the entropy bounded data did not have a significant effect on the result, see Figures 4.16 and 4.17.

The fact that the machine with the highest fitness had a small number of states (four-states) has an obvious explanation. In Chapter 2 a two-state machine for calculating GC-content is described and has the capability of perfectly classifying DNA strings into percent GC-content. Therefore this result is not a surprising one.

Figure 4.16: Mean fitness scores by generation for GC-content data.

Figure 4.17: 95% confidence intervals and best-of-run fitness for GC-content data.

As the number of mutations did not have a significant effect on the results of the GC-content classification, the cross-validation results will only be discussed in terms of number of states. The cross-validation graphs representing four states, Figures 4.18 and 4.19, indicates that the majority of the machines (the cluster of points in the upper right corner) cross-validated very well with only a few machines that appear to have memorized the data (the trail of points beneath the cluster). For state counts of twelve (Figures 4.20 and 4.21) and twenty (Figures 4.22 and 4.23), the data becomes increasingly more spread out indicating that these larger state counts did not cross-validate as well as the four state parameter. The data points that have a high training result and a low cross-validation

result represent those machines that memorized the training data instead of finding the general patterns that separate the two percentages of GC-content data; these machines are said to be *over-trained*. Once again, the parameter settings appear to have had a significant impact on performance.

Figure 4.18: Cross-validation for GC-content data, 4-states, 1 mutation.

Figure 4.19: Cross-validation for GC-content data, 4-states, 2 mutations.

Figure 4.20: Cross-validation for GC-content data, 12-states, 1 mutation.

Figure 4.21: Cross-validation for GC-content data, 12-states, 2 mutations.

Figure 4.22: Cross-validation for GC-content data, 20-states, 1 mutation.

Figure 4.23: Cross-validation for GC-content data, 20-states, 2 mutations.

4.1.3 Stochastic Fitness Experiments

For this experiment, side effect machines were first evolved to classify the two *full* data sets that contain and do not contain a ten character motif. Next, the side effect machines were evolved to classify a randomly chosen partial subset of the two *full* data sets each time an evolutionary run was performed to attempt to reduce susceptibility of memorizing the data. In Figures 4.25 and 4.27, it can be seen that there was not a vast difference in fitness value based on parameter values. On average, the *full* data set had slightly better results with twelve states, two mutations and the *partial* data set had slightly better results with

twenty-eight states, two mutations. Since the motif in the DNA data strings is 10-characters long, a larger-state side effect machine achieving higher fitness is an anticipated result. The selection of the *partial* data set for each evolutionary run had a significant impact on cross-validation results.

Figure 4.24: Mean fitness scores by generation for *full* data set.

Figure 4.25: 95% confidence intervals and best-of-run fitness for *full* data set.

Figure 4.26: Mean fitness scores by generation for *partial* data set.

Figure 4.27: 95% confidence intervals and best-of-run fitness for *partial* data set.

The cross-validation results for the *full* data set is shown in Figures 4.28, 4.29, 4.30, 4.31, 4.32, and 4.33 and *partial* data set results are shown in Figures 4.34, 4.35, 4.36, 4.37, 4.38, and 4.39. Figures for the *full* data set indicate a low cross-validation result as no data points come close to the $y=x$ line and very few data points fall between a cross-validation value of 0.7 and 0.8. This suggests that the *full* data set is memorizing the data set as opposed to generalizing. Figures for the *partial* data set indicate a higher cross-validation result as many data points lie along and above the $y=x$ line and many more points fall between a cross-validation value of 0.7 and 0.8. This suggests that the *partial* data set is generalizing the data instead of the undesirable result of memorizing the data.

Figure 4.28: Cross-validation for *full* data set, 12-states, 1 mutation.

Figure 4.29: Cross-validation for *full* data set, 12-states, 2 mutations.

Figure 4.30: Cross-validation for *full* data set, 20-states, 1 mutation.

Figure 4.31: Cross-validation for *full* data set, 20-states, 2 mutations.

Figure 4.32: Cross-validation for *full* data set, 28-states, 1 mutation.

Figure 4.33: Cross-validation for *full* data set, 28-states, 2 mutations.

Figure 4.34: Cross-validation for *partial* data set, 12-states, 1 mutation.

Figure 4.35: Cross-validation for *partial* data set, 12-states, 2 mutations.

Figure 4.36: Cross-validation for *partial* data set, 20-states, 1 mutation.

Figure 4.37: Cross-validation for *partial* data set, 20-states, 2 mutations.

Figure 4.38: Cross-validation for *partial* data set, 28-states, 1 mutation.

Figure 4.39: Cross-validation for *partial* data set, 28-states, 2 mutations.

As can be seen in Figures 4.40, 4.41, 4.42, 4.43, 4.44, and 4.45 where the *partial* and *full* data set cross-validation results are superimposed into the same graphs, the + symbols representing the *partial* data set in each figure are always closer to the $y=x$ line than the x symbols that represent the *full* data set. These figures show that the *partial* data set cross-validated better than the *full* data set, confirming the hypothesis that randomly selecting a *partial* subset of the *full* data set would help the side effect machines to generalize as opposed to memorizing the data.

Figure 4.40: Cross-validation for *full* & *partial* data sets, 12-states, 1 mutation.

Figure 4.41: Cross-validation for *full* & *partial* data sets, 12-states, 2 mutations.

Figure 4.42: Cross-validation for *full* & *partial* data sets, 20-states, 1 mutation.

Figure 4.43: Cross-validation for *full* & *partial* data sets, 20-states, 2 mutations.

Figure 4.44: Cross-validation for *full* & *partial* data sets, 28-states, 1 mutation.

Figure 4.45: Cross-validation for *full* & *partial* data sets, 28-states, 2 mutations.

4.2 Results and Discussion for Biological Data

The results for the PCR primer data sets, mitochondrial DNA data sets, and intron and exon DNA data sets are stated and explained in the following three sections.

4.2.1 PCR Primer DNA Experiments

The best fitness result on the primer data was 0.788 obtained with a 24-state side effect machine with one mutation. This machine cross-validated with a best Rand index of 0.772.

The worst side effect machine had a Rand index of 0.632 and a cross-validation result of 0.506 which suggests robust generalization is possible but does not always occur.

There were a total of 20 experiments done for the training and test data, multiples of four states from 4 to 40 with 1 and 2 mutations as described in Chapter 3. Only states sizes 12, 16, 20 and 24 cross-validated well, consequently, only those state results are shown in Figures 4.46, 4.47, 4.48, and 4.49.

Figure 4.46: Mean fitness scores by generation for primer data, 12-states.

Figure 4.47: Mean fitness scores by generation for primer data, 16-states.

Figure 4.48: Mean fitness scores by generation for primer data, 20-states & 24-states.

Figure 4.49: 95% confidence intervals and best-of-run fitness for primer data.

Single-point mutation consistently scored better than two-point mutation indicating that once a mode was located, the optima of that mode was easily found with a local search of one mutation. Also, there is a positive correlation between the number of states and the Rand index; as the number of states increase, the Rand index also increases. This is also the trend for all 20 experiments that were run with the training data. The poor cross-validation results on the test data for states 4, 8, 28, 32, 36, and 40 suggests that the presence of very few or very many states results in an over-trained machine more than a moderate number of states. The high fitness result for a 24-state machine can be

attributed to the fact that primers typically have a length of about 20 bases and rarely exceed a length of 30 bases [7].

4.2.2 Mitochondrial DNA Experiments

On average, the best fitness result for the mitochondrial data is an 8-state side effect machine. The Rand index peaks at 8-states and gradually decreases as state size grows larger. There is almost no correlation between number of mutations and Rand index as both 1- and 2-point mutations resulted in very similar fitness values. Also, there is a very small difference between the machine with the best average and the worst average Rand index as they only differ by 0.02 on the Rand index axis. Thus, state sizes did not seem to have a significant effect on performance.

As explained in Chapter 3, cross-validation was not applied to the mitochondrial data as test data was not available and the goal of this experiment was to see if side effect machines could learn how to classify mitochondrial DNA.

Figure 4.50: 95% confidence intervals and best-of-run fitness for mitochondrial data, 1 mutation.

Figure 4.51: 95% confidence intervals and best-of-run fitness for mitochondrial data, 2 mutations.

Figure 4.52: Mean fitness scores by generation for mitochondrial data, 1 mutation.

Figure 4.53: Mean fitness scores by generation for mitochondrial data, 2 mutations.

4.2.3 Intron and Exon DNA Experiments

The best average fitness result for the intron and exon experiments was a 40-state, single-point mutation machine. There was very little difference between state machines of differing sizes with the best and worst average Rand indices only differing by 0.05. In general, a single-point mutation scored a higher fitness than a 2-point mutation.

As explained in Chapter 3, cross-validation was not applied to the intron and exon data sets, like the mitochondrial data, as test data was not available. Similarly to the experimentation with the mitochondrial data sets, the goal of the intron and exon DNA

experiment was to see if side effect machines could learn how to classify this type of data set.

Figure 4.54: 95% confidence intervals and best-of-run fitness for intron & exon data, 1 mutation.

Figure 4.55: 95% confidence intervals and best-of-run fitness for intron & exon data, 2 mutations.

Figure 4.56: Mean fitness scores by generation for intron & exon data, 1 mutation.

Figure 4.57: Mean fitness scores by generation for intron & exon data, 2 mutations.

Chapter 5

Conclusion and Future Direction

5.1 Conclusions

While estimates of good population size and number of mutations remains somewhat consistent between the synthetic and biological data sets, the best number of states did not. A population size of 10 and single-point mutation generally scored higher Rand indices with both synthetic and biological data, while number of states differed within and between synthetic and biological data.

Side effect machine state sizes that produce a high fitness result differ greatly depending on the data set being tested. For the GC-content data, a small side effect machine was found to be effective in classifying the data sets. As explained in Chapter 2, a two-state machine can perform the task as there are two percentages of concern, GC and AT. For the 10-character motif data, a larger side effect machine was found to be effective in classifying the data sets. In Chapter 2, the tag detector side effect machine can perform the task of recognizing a motif in the sequence. A larger state machine was found to be effective as the

data that was to be classified had a motif that was 10-characters long. For the PCR primer data, a larger side effect machine was also effective in correct classification. Primers typically have a length of about 20 bases and rarely exceed a length of 30 bases [7]. These primer length bounds easily explain the reason for a larger state side effect machine having a better performance. An overall conclusion relating data sets to number of states is that the more complex the data set, the larger the required state size of the side effect machine. Despite differing parameter results between data sets, both synthetic and biological data sets showed robust generalization through cross-validation, which is an extremely promising result. Side effect machines are clearly able to learn without memorizing both synthetic and biological data.

5.2 Future Directions

5.2.1 Ignorance Side Effect Machines

In an attempt to filter out unnecessary noise caused by the presence of states in the side effect machines that are of little importance to a particular substring search, the concept of *ignorance side effect machines* was created. This ignorance of states results in fewer dimensions in the feature vector and the resulting k -means cluster of these vectors. This idea was implemented using a binary *use me /don't use me* mutation on the side effect machine array, see Figure 5.1.

	C	G	A	T	1/0
0:	2,	1,	2,	0,	0
1:	0,	2,	3,	3,	1
2:	2,	0,	3,	0,	1
3:	0,	3,	3,	2,	0

Figure 5.1: Example of an ignorance side effect machine as a linear array.

A preliminary study was completed where ignorance side effect machines were trained on the fifth synthetic data set with a maximum entropy of 1.8 and on the PCR primer data set. The synthetic data set was run with state counts of 4, 12, 20, and 28 with 1, 2, and 3 mutations, population size 10 and up to 10,000 mating events. The PCR primer data set was run with state counts of 4, 8, 12, 16, 20, 24, 28, 32, 36, and 40 with 1 and 2 mutations, population size 10 and up to 20,000 mating events.

The results from the fifth synthetic data set showed that the problem was perhaps too easy for ignorance side effect machines to classify as there was no difference from the fitness values obtained using side effect machines without the binary mutation. For the PCR primer data, the classification problem was too hard for ignorance side effect machines to classify as all fitness resulted in over-training. In this case, ignorance prevents fitness as PCR primers, as discussed in Section 5.1, require the use of a large numbered state side effect machine. Given that selecting features is valuable in most machine learning, sub-selection of states will probably prove valuable for *some* sequence classification problems. Ignorance side effect machines appear in Section 5.2 because there is a need for them to be tested on more biological sequences.

5.2.2 k -Nearest Neighbor Algorithm

The k -nearest neighbor algorithm is used as a method of classifying data based on closest training examples in the feature space, based on a distance metric (usually Euclidean distance) [29]. A data point in the feature space is classified by a majority rule of its k -nearest neighbors, see Figure 5.2.

Figure 5.2: k -nearest neighbor example. The $\square$ in the center of the figure represents a point to be classified and the $\star$'s and $\diamond$'s represent two classes of the training data. If k is chosen to be 3, the 3 closest points to the $\square$ are chosen (represented by the inner circle) and the $\square$ is classified to whichever class has the majority in that circle - in this example, if $k=3$ then the $\square$ is classified with the $\diamond$'s. Similarly, if $k=5$ (represented by the outer circle) the $\square$ is classified with the $\star$'s.

This algorithm can be implemented in place of the k -means clustering algorithm as a contrasting method of classification. The main reason for contrast between k -means clustering and the k -nearest neighbor algorithm is that k -means clustering requires convex clusters due to the nature of the algorithm, whereas the k -nearest neighbor algorithm does not. Thus, the two classifying algorithms are each suitable for different types of problems.

5.2.3 Contrasting Data Targets

Due to the vast nature of the genome, there is a wide array of data targets for which side effect machines can be utilized to classify. This thesis addresses only a few of many data targets. The following describes a few more useful data targets in which side effect machines could be used to classify.

Non-coding RNA (ncRNA)

Non-coding RNA is classified as a functional RNA molecule that is not translated into a protein. ncRNA genes have significant biological functions and are highly abundant. The following describes two examples of ncRNA.

A stem-loop (or hairpin loop) nucleotide structure is commonly found in RNA. This structure forms when two regions in the same molecule base-pair to assemble a double helix with an unpaired loop [38]. This only comes about when there is a RNA palindrome within the single-stranded nucleotide molecule. A RNA palindrome can be described by two substrings of one sequence where the first nucleotide base of the first substring pairs with the last base of the second substring, the second base of the first substring pairs with the second-last base of the second substring and so on, causing the strand to fold back on itself through base-pair bonds. Stem-loop structures are one of the key building blocks of RNA secondary structure.

Figure 5.3: Stem-loop structure.

Small interfering RNA (siRNA) is another type of non-coding RNA. They are small double-stranded RNA molecules that are 20 to 25 nucleotides in length that inhibit gene expression [38]. Since many disease processes depend on the activity of several genes, the use of siRNAs could prove useful in disease treatment.

An obvious area for future work is the application of side effect machines to other kinds of biological data, these include the stem-loop nucleotide structure and small interfering RNA as well as targets of opportunity provided by biological collaborators.

References

- [1] B. Alberts, A. Johnson, J. Lewis, K. Roberts, and P. Walter, *Molecular biology of the cell*, 4 ed., Garland Science, New York, NY, 2002.
- [2] D. Ashlock, *Evolutionary computation for modeling and optimization*, Springer, New York, NY, 2006.
- [3] D. Ashlock and E. Warner, *Classifying synthetic and biological DNA sequences with side effect machines*, IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology, 2008 (2008), 22–29.
- [4] W. Ashlock, *Using very small population sizes in genetic programming*, 2006 IEEE Congress on Evolutionary Computation, IEEE Press, 2006, pp. 319–326.
- [5] G. E. P. Box, *Evolutionary operation: A method for increasing industrial productivity*, Applied Statistics **6** (1957), no. 2, 81–101.
- [6] H. J. Bremermann, *Optimization through evolution and recombination*, Self-Organizing Systems (M. C. Yovits, G. T. Jacobi, and G. D. Goldstein, eds.), Spartan Books, Washington, D.C., 1962.

- [7] T. A. Brown, *Gene cloning and DNA analysis: An introduction*, 5 ed., Blackwell Publishing, Oxford, UK, 2006.
- [8] D. Corne, A. Meade, and R. Sibly, *Evolving core promoter signal motifs*, Proceedings of the 2001 Congress on Evolutionary Computation (Seoul, South Korea), vol. 2, IEEE Press, 2001, pp. 1162–1169.
- [9] D. B. Fogel, *Evolutionary computation: The fossil record*, IEEE Press, Piscataway, NJ, 1998.
- [10] G. B. Fogel and D. W. Corne, *Evolutionary computation in bioinformatics*, Morgan Kaufmann Publishers, San Francisco, CA, 2003.
- [11] L. J. Fogel, *Autonomous automata*, Industrial Research Magazine **4** (1962), 14–19.
- [12] R. M. Friedberg, *A learning machine: Part I*, IBM Journal of Research and Development **2** (1958), no. 1, 2–13.
- [13] R. M. Friedberg, B. Dunham, and J. H. North, *A learning machine: Part II*, IBM Journal of Research and Development **3** (1959), no. 7, 282–287.
- [14] B. Henrissat, *A classification of glycosyl hydrolases based on amino acid sequence similarities*, Biochemistry **280** (1991), 309–316.
- [15] H. Hirsh and M. Noordewier, *Using background knowledge to improve inductive learning*, Proceedings of the 10th Conference on Artificial Intelligence for Applications (Los Alamitos, CA), IEEE Press, 1994, pp. 351–357.

- [16] J. H. Holland, *Outline for a logical theory of adaptive systems*, Journal of the Association for Computing Machinery **3** (1962), 297–314.
- [17] H. R. Horton, L. A. Moran, R. S. Ochs, J. D. Rawn, and K. G. Scrimgeour, *Principles of biochemistry*, 3 ed., Prentice-Hall, Upper Saddle River, NJ, 2002.
- [18] N. B. Ishiguro, M. Miya, J. G. Inoue, and M. Nishida, *Sundasalanx (Sundasalangidae) is a progenetic clupeiform, not a closely related group of salangids (Osmeriformes): Mitogenomic evidence*, Journal of Fish Biology **67** (2005), 561–569.
- [19] L. T. Kirby, *DNA fingerprinting: An introduction*, Stockton Press, New York, NY, 1990.
- [20] N. Lane, *Power, sex, suicide: Mitochondria and the meaning of life*, Oxford University Press, New York, NY, 2005.
- [21] D. Loewenstern, H. Hirsh, P. Yianilos, and M. Noordewier, *DNA sequence classification using compression-based induction*, DIMACS Technical Report 95–04, Rutgers University, New Brunswick, NJ, April 1995.
- [22] R. L. Raaum, K. N. Sterner, C. M. Noviello, C. B. Stewart, and T. R. Disotell, *Catarrhine primate divergence dates estimated from complete mitochondrial genomes: Concordance with fossil and nuclear DNA evidence*, Journal of Human Evolution **48** (2005), no. 3, 237–257.
- [23] Q. Ma, J. T. L. Wang, D. Shasha, and C. H. Wu, *DNA sequence classification via an expectation maximization algorithm and neural networks: A case study*, IEEE Transactions on Systems, Man, and Cybernetics **31** (2001), no. 4, 468–475.

- [24] A. Meade, D. Corne, and R. Sibly, *Discovering patterns in microsatellite flanks with evolutionary computation by evolving discriminatory DNA motifs*, Proceedings of the 2002 Congress on Evolutionary Computation (Honolulu, HI, USA), IEEE Press, 2002, pp. 1–6.
- [25] A. Meade, R. Sibly, and D. Corne, *Applying evolutionary computation to understanding the insertion behavior of LINE-1 retrotransposons in human DNA*, Proceedings of the 2000 Congress on Evolutionary Computation (La Jolla, CA, USA), vol. 2, IEEE Press, 2000, pp. 836–843.
- [26] M. Miya, H. Takeshima, H. Endo, N. B. Ishiguro, and J. G. Inoue, *Major patterns of higher teleostean phylogenies: A new perspective based on 100 complete mitochondrial DNA sequences*, Molecular Phylogenetic Evolution **26** (2003), 121–138.
- [27] C.R. Newton and A. Graham, *PCR*, BIOS Scientific Publishers Limited, Oxford, UK, 1994.
- [28] T. Palys, L.K. Nakamura, and F. M. Cohan, *Discovery and classification of ecological diversity in the bacterial world: The role of DNA sequence data*, International Journal of Systematic Bacteriology **47** (1997), no. 4, 1145–1156.
- [29] A. N. Papadopoulos and Y. Manolopoulos, *Nearest neighbor search: A database perspective*, Springer, New York, NY, 2005.
- [30] C. Pridgeon and D. Corne, *Hypermotifs: Novel discriminatory patterns for nucleotide sequences and their application to core promoter prediction in eukaryotes*, Proceedings

- of the 2005 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology, 2005 (Los Alamitos, CA), IEEE Press, 2005, pp. 1–7.
- [31] I. Rechenberg, *Cybernetic solution path of an experimental problem*, Library Translation 1122, Royal Aircraft Establishment, Farnborough, UK, August 1965.
- [32] H. Schwefel, *Experimentelle Optimierung einer Zweiphasendüse, part I*, Technical Report No. 35 of the Project MHD–Staustahlrohr 11.034/68, AEG Research Institute, Berlin, October 1968.
- [33] D. Silvestre, M. Dowton, and M. C. Arias, *The mitochondrial genome of the stingless bee *Melipona bicolor* (Hymenoptera, Apidae, Meliponini): Sequence, gene organization, and a unique tRNA translocation event conserved across the tribe Meliponini*, Genetic and Molecular Biology **31** (2008), no. 2, 451–460.
- [34] K. E. Slack, C. M. Jones, T. Ando, G. L. Harrison, and R. E. Fordyce, *Early penguin fossils, plus mitochondrial genomes, calibrate avian evolution*, Molecular Biological Evolution **23** (2006), no. 6, 1144–1155.
- [35] J. H. van Lint and R. M. Wilson, *A course in combinatorics*, 2 ed., Cambridge University Press, Cambridge, UK, 2006.
- [36] J. T. L. Wang, S. Rozen, B. A. Shapiro, D. Shasha, Z. Wang, and M. Yin, *New techniques for DNA sequence classification*, Journal of Computational Biology **6** (1999), no. 2, 209–218.

- [37] K. Y. Yeung and W. L. Ruzzo, *Details of the adjusted Rand index and clustering algorithms. Supplement to the paper “An empirical study on principal component analysis for clustering gene expression data”*, Bioinformatics **17** (2001), 763–774.
- [38] S. Ying, *MicroRNA protocols*, Humana Press, Totowa, NJ, 2006.

Appendix A

Code

A.1 Side Effect Machine Evolver Code

```
/* This is the first implementation of a side effect machines. It
 * uses the Rand index as a fitness function to train FSMs to
 * classify DNA sequences.
 *
 * Compile line (your computer/the cluster)
 * g++ -lm -O3 EvoSEM.cpp Kmeans.cpp sem.cpp
 * icc -fast EvoSEM.cpp Kmeans.cpp sem.cpp
 *
 * Put a.out and the data file you're using in their own directory
 * and then run a.out. This will generate a best.sem of the best
 * machine from each run and 100 run files. The run files have lines
 * like:
 *
 * 0.786479 0.00862875 0.0880485 0.956611
 *      mean      error      std. dev.  best
 *
 *      bar
 */

//Include the relevant includes
#include <iostream>
#include <fstream>
#include <cmath>
#include <cstdlib>
#include <cstdio>
#include <ctime>
#include <cstring>

using namespace std;
```

```

#include "sem.h"
#include "Kmeans.h"

//data file to work on
#define datafile "Training.dat"

//number of sequences in the data set
#define SS 1000

//number of sequence classes
#define NSC 2

//population size
#define popsize 10

//tournament size
#define tsize 7

//Number of states in the finite state machine
#define VARS 12

//Number of mutations per new machine
#define NMM 1

//maximum number of mating events
#define M 10000

//reporting intervals
#define RI 100

//number of runs to perform
#define R 100

aut pop[popsize];           //the population
double fit[popsize];        //fitness array
double bestfit;             //best fitness so far
int dx[popsize];           //sorting index
int *data[SS];              //DNA data
int clss[SS];               //known class data
int slen[SS];               //sequence lengths
double **sig;               //buffer for signature evaluation

void readdata();             //read the data we are training on
void initialize();           //initialize full algorithm, e.g. population storage
void initpop();              //initialize the population
double fitness(aut &a);       //compute the fitness of and automata
void mate();                 //perform a mating event
void report(ostream &aus);    //interim statistical reporting
void reportbest(ostream &aus); //report best machine at end

main(){

    int r,m;                 //run and mating event index variables
    fstream resf1;           //results file

```

```

double mu,sg,delta; //statistical variables
int failures; //number of timeouts
double N,V; //statistical varliables
char buf[60]; //string construction buffer
double p,q; //probability of failure
fstream stat,crit; //statistics and best machines

readdata(); //read in the data set
initialize(); //allocate population storage, etc.

crit.open("best.sem",ios::out);
for(r=0;r<R;r++){//loop over runs
    sprintf(buf,"run%d.dat",r);
    stat.open(buf,ios::out);
    initpop();
    m=0; //start mating even counter at zero
    while(m<M){//loop until failure or success
        if(m%RI==0){
            cout << r << " " << m << " ";
            report(stat);
        }
        mate(); //do a amting event
        m++; //increment counter
    }
    cout << r << " " << m << " ";
    report(stat);
    stat.close();
    reportbest(crit);
}
crit.close();
}

int CVD(char ch){
    switch(ch){
        case 'c':
        case 'C': return(0);
            break;
        case 'g':
        case 'G': return(1);
            break;
        case 'a':
        case 'A': return(2);
            break;
        case 't':
        case 'T': return(3);
            break;
    }
    cout << "BAD DNA BASE!!! " << ((int) ch) << endl;
    return(4);
}

void readdata(){

```

```

fstream inp;
char buf[1000];
int i,j,k,cv;

i=0;
inp.open(datafile,ios::in);
inp.getline(buf,999);
while(strlen(buf)>3){
    //cout << buf << endl;
    class[i]=atoi(buf);
    //cout << "Class: " << class[i] << endl;
    j=0;
    while(buf[j]!=' '){j++;}
    j++;
    slen[i]=strlen(buf)-j;
    //cout << "Length: " << slen[i] << endl;
    data[i]=new int[slen[i]];
    for(k=0;k<slen[i];k++)data[i][k]=CVD(buf[j++]);
    i++;
    inp.getline(buf,999);
}
}

void initialize(){//initialize full algorithm, e.g. population storage

int i;

srand48(31207813); //seed the random number generator
for(i=0;i<popsize;i++)pop[i].create(VARS);

//create the signature buffer
sig=new double*[SS];
for(i=0;i<SS;i++)sig[i]=new double[VARS];

}

void initpop(){//initialize the population

int i,j;

for(i=0;i<popsize;i++){
    dx[i]=i; //make sure the sorting index is intact
    pop[i].recreate();
    fit[i]=fitness(pop[i]);
}
}

/* This is the experimental section of the code. We run each sequence
through the FSM. It records the number of times each state is
visited. We then cluster the visitation vectors and see how well the
machine learns the correct arrangement of DNA sequences. The
clustering algorithm is K-means and the assessment is the Rand index.
*/

```

```

double fitness(aut &a){//

int classn[SS];           //class number from K-means clustering
int i,j,cnt;              //loop indices
int cv[VARs];             //reporting variable
int r,s;                  //Rand index comparison variables
double rv,dv;             //return value and divisor
double ttl;               //total

    //cout << "Starting fitness evaluation" << endl;

    //gather the side effect signatures
    for(i=0;i<SS;i++){//loop over sequences
        a.reset(); //reset the automata
        for(j=0;j<slen[i];j++)a.run(data[i][j]); //run the sequence data
        a.report(cv); //get the state-use count vector
        ttl=0.0;
        for(j=0;j<VARs;j++){
            sig[i][j]=((double)cv[j]); //store it as real data
            ttl+=sig[i][j]; //get the total length
        }
        for(j=0;j<VARs;j++)sig[i][j]/=ttl; //normalize the length
    }

    //perform the K-means clustering step
    Kmeans(sig,SS,VARs,NSC,classn);

    //compute the Rand index
    cnt=0;
    for(i=0;i<SS-1;i++)for(j=i+1;j<SS;j++){
        r=(classn[i]==classn[j]);
        s=(classn[i]==classn[j]);
        //cout << i/60 << " " << j/60 << " " << classn[i] << " " << classn[j];
        if(r==s)cnt++;
        //cout << " " << cnt << endl;
    }

    dv=((double)SS);
    rv=((double)cnt);
    rv=(2*rv)/dv/(dv-1);

    //cout << rv << " " << endl;

    return(rv);
}

void mate(){//perform a mating event

int i,j;                //loop indices
int sw;                 //swap variable
int flag;               //sorting flag
int cp1,cp2;            //crossover points

```

```

for(i=0;i<tsize;i++){//swap a tournament to the front of the sorting order
    j=lrand48()%popsize;
    sw=dx[i];dx[i]=dx[j];dx[j]=sw;
}

//reevaluate the fitness of the tournament members
for(i=0;i<tsize;i++){
    fit[dx[i]]=fitness(pop[dx[i]]);
}

//sort the tournament so that the two worst fitnesses are first
do {
    flag=0;
    for(i=0;i<tsize-1;i++)if(fit[dx[i]]>fit[dx[i+1]]){
        sw=dx[i];dx[i]=dx[i+1];dx[i+1]=sw;
        flag=1;
    }
}while(flag);

//perform reproduction and crossover
pop[dx[0]].copy(pop[dx[tsize-2]]);
pop[dx[1]].copy(pop[dx[tsize-1]]);
pop[dx[0]].tpc(pop[dx[1]]);

//perform mutations
for(i=0;i<NMM;i++){
    pop[0].mutate();
    pop[1].mutate();
}

//assess fitness
fit[dx[0]]=fitness(pop[dx[0]]);
fit[dx[1]]=fitness(pop[dx[1]]);
}

void report(ostream &aus){//interim statistical reporting

double mu,sg,bs; //mean, variance, best
int i; //loop index

mu=sg=bs=0.0;
for(i=0;i<popsize;i++){
    mu+=fit[i];
    sg+=(fit[i]*fit[i]);
    if(fit[i]>bs)bs=fit[i];
}
mu/=((double)popsize);
sg/=((double)popsize);
sg-=(mu*mu);
if(sg<0)sg=0.0; else sg=sqrt(sg);
aus << mu << " " << 1.96*sg/sqrt((double)popsize)
    << " " << sg << " " << bs << endl;
cout << mu << " " << 1.96*sg/sqrt((double)popsize)

```

```

        << " " << sg << " " << bs << endl;
    }

    void reportbest(ostream &aus){//report best machine at end

    int i,bdx;
    double bs;

    bs=0.0;
    for (i=0;i<popsize;i++)if (fit[i]>bs){
        bs=fit[i];
        bdx=i;
    }

    aus << bs << " fitness" << endl;
    pop[bdx].write(aus);
}

```

A.2 Side Effect Machine Population Creator Code

```
/* Side effect machine class of finite state automata
 *
 * code file
 */

#include <iostream>
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <cctype>
#include <cmath>

#include "sem.h"

//constructors and destructors
aut::aut(){//create an empty automata

    blank();
}

aut::aut(int nv){//create an automata with nv states

    n=0;
    create(nv);
}

aut::aut(const aut &other){//copy constructor

    int i,j;

    n=other.n;
    cur=other.cur;
    cntr=new int[n];
    states=new int*[n];
    for(i=0;i<n;i++){
        states[i]=new int[TR];
        for(j=0;j<TR;j++)states[i][j]=other.states[i][j];
        cntr[i]=other.cntr[i];
    }
}

aut::~aut(){//deallocate the automata

    clear();
}

//management routines
void aut::blank(){//create an empty automata

    n=0;
}
```



```

void aut::clear(){//clear the automata

    if(n){
        for(int i=0;i<n;i++)delete [] states[i];
        delete [] states;
        delete [] cntr;
    }
    n=0;
}

void aut::create(int nv){//create an automata with nv states

    if(n)clear();
    n=nv;
    states=new int*[n];
    for(int i=0;i<n;i++)states[i]=new int[TR];
    cntr=new int[n];
    recreate();
}

void aut::recreate(){//randomize an allocated automata

    int i,j;

    if(n){//failsafe the recreation of empty automata
        for(i=0;i<n;i++){//loop over states
            for(j=0;j<TR;j++)states[i][j]=lrand48()%n; //make transitions
        }
    }
}

void aut::copy(aut &other){//make a copy of another automata

    int i,j;

    if(n!=other.n)create(other.n);
    cur=other.cur;
    for(i=0;i<n;i++){
        for(j=0;j<TR;j++)states[i][j]=other.states[i][j];
        cntr[i]=other.cntr[i];
    }
}

//use routines
void aut::reset(){//set state to zero and clear counters

    cur=0; //set current state to zero
    for(int i=0;i<n;i++)cntr[i]=0; //zero the counters
    cntr[0]=1; //record the starting state
}

void aut::run(int v){//run on input value v

```

```

    cur=states[cur][v]; //make the state transition
    cntr[cur]++; //record the entry into the state
}

void aut::report(int *rp){//report the state usage vectors

    //not that rp is presumed to be long enough
    for(int i=0;i<n;i++)rp[i]=cntr[i];
}

//genetic operations
void aut::tpc(aut &other){//perform two point crossover

int i,j,sw,cp1,cp2;

    cp1=rand48()%n; //find first crossover point
    cp2=rand48()%n; //find second crossover point
    if(cp1>cp2){sw=cp1;cp1=cp2;cp2=sw;} //order them
    for(i=cp1;i<=cp2;i++){
        for(j=0;j<TR;j++){//cross over the transitions
            sw=states[i][j];
            states[i][j]=other.states[i][j];
            other.states[i][j]=sw;
        }
    }
}

void aut::mutate(){//perform a transition mutation

    states[lrand48()%n][lrnd48()%TR]=lrnd48()%n;
}

//input and output
void aut::write(ostream &aus){//output method

int i,j;

    aus << n << " states" << endl;
    for(i=0;i<n;i++){
        aus << i << ":";
        aus << states[i][0];
        for(j=1;j<TR;j++)aus << "," << states[i][j];
        aus << endl;
    }
}

void aut::read(istream &inp){//input method

char buf[1000];
int i,j,k;
    inp.getline(buf,999);
    create(atoi(buf));
    for(i=0;i<n;i++){
        inp.getline(buf,999);

```

```

j=0;
while(buf[j]!=':')j++;
for(k=0;k<IR;k++){
    j++;
    states[i][k]=atoi(buf+j);
    while(isdigit(buf[j]))j++;
}
}
}

```

A.3 Side Effect Machine Population Creator Header File

```
/* Side effect machine class of finite state automata
 * header file
 */
#ifndef _SEM.H
#define _SEM.H

using namespace std;

//number of transitions out per state
#define TR 4

class aut { //side effect machine class

public:

    //constructors and destructors
    aut(); //create an empty automata
    aut(int nv); //create an automata with nv states
    aut(const aut &); //copy constructor
    ~aut(); //deallocate the automata

    //management routines
    void blank(); //create an empty automata
    void clear(); //clear the automata
    void create(int nv); //create an automata with nv states
    void recreate(); //randomize an allocated automata
    void copy(aut &other); //make a copy of another automata

    //use routines
    void reset(); //set state to zero and clear counters
    void run(int v); //run on input value v
    void report(int *rp); //report the state usage vectors

    //genetic operations
    void tpc(aut &other); //perform two point crossover
    void mutate(); //perform a transition mutation

    //input and output
    void write(ostream &aus);
    void read(istream &inp);

private:

    int n; //number of states
    int cur; //current state
    int **states; //state vector
    int *cntr; //counter vector
};
#endif /* _SEM.H */
```

A.4 k -Means Clustering Code

```
/* Primitive k-means clustering for use in multiclustering
*/

/*
* The routine Kmeans accepts a data array data[N][dim] of N points with
* dim coordinates. I then performs a k-means clustering on them with
* k clusters and returns the cluster assignments for points as numbers
* 0..k-1 in the array class[N]
*/

#include <iostream>
#include <cstdlib>

#include "Kmeans.h"

void Kmeans(double **data,int N,int dim,int k,int *classn){

    TruncatedKmeans(data,N,dim,k,0,classn);
}

void TruncatedKmeans(double **data,int N,int dim,int k,int m,int *classn){

    int i,j,l;                //loop index variables
    int flag;                 //flag for detecting cluster assignment change
    double cen[k][dim];       //holds cluster centers
    int cz[k];                //current cluster sizes
    double q,cd,mind;          //distance buffers used in moving points
    int dx;                   //index of closest center for moving points
    int exc[k];                //exclusion index for initial centers
    int cnt;                   //number of K-means loops

    //debugging variables
    int MV;

    //make initial random cluster assignments
    for(i=0;i<N;i++)classn[i]=lrand48()%k;
    for(i=0;i<k;i++){//choose random points as initial cluster centers
        do{//select a point not selected before
            flag=0;
            j=lrand48()%N;
            for(l=0;l<i;l++)if(j==exc[l]) flag=1;
        } while(flag);
        //cout << j << " ";
        exc[i]=j;
        for(l=0;l<dim;l++)cen[i][l]=data[j][l];
    }
    //cout << endl;
    //ready to start k-means loop
    flag=1;
    cnt=0;
    while(flag){
        //centers are now known, we can reassign the points
```

```

flag=0; //if no points move we're done
MV=0;

for(i=0;i<N;i++){//for each point
    for(j=0;j<k;j++){//for each center

        cd=0.0;
        for(l=0;l<dim;l++){//compute squared distance from point to center
            q=data[i][l]-cen[j][l];
            cd+=q*q;
        }

        if((j==0)||((cd<mind))){//if distance is first of better
            dx=j; //assign it
            mind=cd;
        }
    }

    if(classn[i]!=dx){//point moved
        classn[i]=dx; //assign new class
        flag=1; //flag it as "movement happened"
        MV++;
    }
}

//for(i=0;i<k;i++)cout << cz[i] << " ";
//cout << endl;
//cout << MV << " points moved." << endl;

//compute current cluster centers based on current assignment
for(i=0;i<k;i++){//zero initial centers and cluster sizes
    for(j=0;j<dim;j++)cen[i][j]=0.0;
    cz[i]=0;
}

for(i=0;i<N;i++){//for all points
    cz[classn[i]]++; //note the increase in class size
    for(j=0;j<dim;j++)cen[classn[i]][j]+=data[i][j]; //sum to center
}
//divide summed centers by class sizes
for(i=0;i<k;i++)for(j=0;j<dim;j++)cen[i][j]/=((double)cz[i]);
cnt++;
if(cnt==m)flag=0; //impose truncation
if(cnt>5*N)flag=0; //force termination in infinit loop
}
}

```

A.5 k -Means Clustering Header File

```
/* Primitive k-means clustering for use in multiclustering
 */

using namespace std;

/*
 * The routine Kmeans accepts a data array data[N][dim] of N points with
 * dim coordinates. I then performs a k-means clustering on them with
 * k clusters and returns the cluster assignments for points as numbers
 * 0..k-1 in the array class[N]
 */
void Kmeans(double **data,int N,int dim,int k,int *classn);

/**
 * Truncated K-means is like k-means save that it takes an additional
 * argument m that limits the number of iterations the k-means algorithm
 * runs. Instead of running to completion it runs until completion OR
 * the iteration limit m.
 */
void TruncatedKmeans(double **data,int N,int dim,int k,int m,int *classn);
```

A.6 Cross-validation Code

```
/* This code is intended to cross validate primer-evolved side effect
 * machines.
 */

//Include the relevant includes
#include <iostream>
#include <fstream>
#include <cmath>
#include <cstdlib>
#include <cstdio>
#include <ctime>
#include <cstring>

using namespace std;

#include "sem.h"
#include "Kmeans.h"

//automata file nam
#define AFN "best.sem"
//number of sequences in the data set
#define SS 1000
//number of sequence classes
#define NSC 2
//Number of states in the finite state machine
#define VARS 20
#define CVset "Crossvalidation.dat"

#define Naut 100

int *data[SS];           //DNA data
aut pop[Naut];           //automata to be cross validated
int cls[SS];             //known class data
int slen[SS];            //sequence lengths
double **sig;            //buffer for signature evaluation
double Of[Naut], Nf[Naut]; //training and crossvalidation fitnesses

void initialize(); //initialize full algorithm, e.g. population storage
void readdata();   //read the data we are crossvalidating with
void readaut();    //read in the best file
double fitness(aut &a); //compute the fitness of an automata

main(){

int i;

initialize();
readdata();
readaut();
for(i=0; i<Naut; i++){
    Nf[i]=fitness(pop[i]);
    cout << Of[i] << " " << Nf[i] << endl;
}
```



```

    }
}

void initialize(){//initialize full algorithm, e.g. population storage

int i;

    srand48(31207813); //seed the random number generator
    //create the signature buffer
    sig=new double*[SS];
    for(i=0;i<SS;i++)sig[i]=new double[VARs];
}

int CVD(char ch){

    switch(ch){
    case 'c':
    case 'C': return(0);
        break;
    case 'g':
    case 'G': return(1);
        break;
    case 'a':
    case 'A': return(2);
        break;
    case 't':
    case 'T': return(3);
        break;
    }
    cout << "BAD DNA BASE!!! " << ((int) ch) << endl;
    return(4);
}

void readdata(){

    fstream inp;
    char buf[1000];
    int i,j,k,cv;

    i=0;
    inp.open(CVset,ios::in);
    inp.getline(buf,999);
    while(strlen(buf)>3){
        //cout << buf << endl;
        class[i]=atoi(buf);
        //cout << "Class: " << class[i] << endl;
        j=0;
        while(buf[j]!=' '){j++;}
        j++;
        slen[i]=strlen(buf)-j;
        //cout << "Length: " << slen[i] << endl;
        data[i]=new int[slen[i]+1];
        for(k=0;k<slen[i];k++)data[i][k]=CVD(buf[j++]);
        i++;
    }
}

```

```

    inp.getline(buf,999);
}
}

void readaut(){

fstream inp;
char dum[60];
int i;

    inp.open(AFN,ios::in);
    for(i=0;i<Naut;i++){
        inp.getline(dum,59);
        Of[i]=atof(dum);
        //cout << dum << endl;
        pop[i].read(inp);
    }
    inp.close();
}

double fitness(aut &a){//

int classn[SS];           //class number from K-means clustering
int i,j,cnt;              //loop indices
int cv[VARs];             //reporting variable
int r,s;                  //Rand index comparison variables
double rv,dv;             //return value and divisor
double ttl;               //total

    //cout << "Starting fitness evaluation" << endl;

    //gather the side effect signatures
    for(i=0;i<SS;i++){//loop over sequences
        a.reset(); //reset the automata
        for(j=0;j<slen[i];j++)a.run(data[i][j]); //run the sequence data
        a.report(cv); //get the state-use count vector
        ttl=0.0;
        for(j=0;j<VARs;j++){
            sig[i][j]=((double)cv[j]); //store it as real data
            ttl+=sig[i][j]; //get the total length
        }
        for(j=0;j<VARs;j++)sig[i][j]/=ttl; //normalize the length
    }

    //perform the K-means clustering step
    Kmeans(sig,SS,VARs,NSC,classn);

    //compute the Rand index
    cnt=0;
    for(i=0;i<SS-1;i++)for(j=i+1;j<SS;j++){
        r=(classn[i]==classn[j]);
        s=(classn[i]==classn[j]);
        //cout << i/60 << " " << j/60 << " " << classn[i] << " " << classn[j];
        if(r==s)cnt++;
    }
}

```

```

    //cout << " " << cnt << endl;
}

dv=((double)SS);
rv=((double)cnt);
rv=(2*rv)/dv/(dv-1);

//cout << rv << " " << endl;

return(rv);
}

```

A.7 GC-Content Data Creator Code

```
/* This code implements a simple randomization to create synthetic
 * DNA with give fractions of GC in the DNA.
 */
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <ctime>
#include <cmath>

using namespace std;

//number of classess to generate
#define NC 2
//number of sequences to generate per class
#define NS 1000
//string minimum and maximum length
#define minLEN 100
#define LEN 200
//GC percentages for the classess
#define pc1 0.55
#define pc2 0.45

main(){

char *fyle[2]={ "Training.dat", "Crossvalidation.dat" };
char *DNA="CGAT"; //conversion string
int i,j,k,n;
int a,b,sw;
fstream aus;
int buf[LEN];

srand48(91207819);
for(n=0;n<2;n++){
    aus.open(fyle[n], ios::out);
    for(i=0;i<NS/2;i++){
        k=((int)(LEN*pc1));
        for(j=0;j<LEN;j++)buf[j]=lrand48()%2; //create random bases
        for(j=k;j<LEN;j++)buf[j]+=2; //shift tail to be AT
        for(j=0;j<3*LEN;j++){
            a=lrand48()%LEN;
            b=lrand48()%LEN;
            sw=buf[a]; buf[a]=buf[b]; buf[b]=sw;
        }
        k=lrand48()%(LEN-minLEN)+minLEN;
        aus << 0 << " ";
        for(j=0;j<k;j++)aus << DNA[buf[j]];
        aus << endl;
    }
    for(i=0;i<NS/2;i++){
        k=((int)(LEN*pc2));
        for(j=0;j<LEN;j++)buf[j]=lrand48()%2; //create random bases
        for(j=k;j<LEN;j++)buf[j]+=2; //shift tail to be AT
```

```

    for (j=0;j<3*LEN;j++){
        a=rand48()%LEN;
        b=rand48()%LEN;
        sw=buf[a]; buf[a]=buf[b]; buf[b]=sw;
    }
    k=rand48()%(LEN-minLEN)+minLEN;
    aus << 1 << " ";
    for (j=0;j<k;j++)aus << DNA[buf[j]];
    aus << endl;
}
aus.close();
}
}

```

A.8 Partial Fitness Evaluation Code

```
/*
 * This copy hacked to use partial fitness evaluation
 *
 * This is the first implementation of a side effect machines. It
 * uses the Rand index as a fitness function to train FSMs to
 * classify DNA sequences.
 *
 * Compile line (your computer/the cluster)
 * g++ -lm -O3 EvoSEM.cpp Kmeans.cpp sem.cpp
 * icc -fast EvoSEM.cpp Kmeans.cpp sem.cpp
 *
 * Put a.out and the data file you're using in their own directory
 * and then run a.out. This will generate a best.sem of the best
 * machine from each run and 100 run files. The run files have lines
 * like:
 *
 * 0.786479 0.00862875 0.0880485 0.956611
 *      mean      error    std. dev.  best
 *              bar
 */

//Include the relevant includes
#include <iostream>
#include <fstream>
#include <cmath>
#include <cstdlib>
#include <cstdio>
#include <ctime>
#include <cstring>

using namespace std;

#include "sem.h"
#include "Kmeans.h"

//data file to work on
#define datafile "Training.dat"

//number of sequences in data set
#define SSF 1000

//upper bound on sequence length
#define UBSL 250

//number of sequences to use in training
#define SS 200

//number of sequence classes
#define NSC 2

//population size
#define popsize 10
```

```

//tournament size
#define tsize 7

//Number of states in the finite state machine
#define VARS 28

//Number of mutations per new machine
#define NMM 2

//maximum number of mating events
#define M 20000

//reporting intervals
#define RI 200

//data refreshment interval
#define DR 40

//number of runs to perform
#define R 100

aut pop[popsize];           //the population
double fit[popsize];        //fitness array
double bestfit;             //best fitness so far
int dx[popsize];            //sorting index
int *alldata[SSF];          //Database of DNA data
int allciss[SSF];           //Database of data classess
int allslen[SSF];           //Database of sequence lengths
int *data[SS];              //DNA data
int clss[SS];               //known class data
int slen[SS];               //sequence lengths
int DX[SSF];                //data sorting index
double **sig;               //buffer for signature evaluation
double **allsig;            //buffer for full signature evaluation

void readdata();             //read the data we are training on
void initialize();           //initialize full algorithm, e.g. population storage
void initpop();              //initialize the population
void initdata();             //subselect a data set from the database
double fitness(aut &a);       //compute the fitness of and automata
double fullfitness(aut &a);   //compute the full fitness of and automata
void mate();                 //perform a mating event
void report(ostream &aus);    //interim statistical reporting
void reportbest(ostream &aus); //report best machine at end

main(){

int r,m;                     //run and mating event index variables
fstream resfyl;              //results file
double mu,sg,delta;          //statistical variables
int failures;                //number of timeouts
double N,V;                  //statistical varliables
char buf[60];                //string construction buffer

```

```

double p,q;           //probability of failure
fstream stat,crit;     //statistics and best machines

readdata();           //read in the data set
initialize();         //allocate population storage, etc.

crit.open("best.sem",ios::out);
for(r=0;r<R;r++){//loop over runs
    sprintf(buf,"run%d.dat",r);
    stat.open(buf,ios::out);
    initdata();       //select an initial data set
    initpop();
    m=0;              //start mating even counter at zero
    while(m<M){//loop until failure or success

        if(n%RI==0){//report at reporting intervals
            cout << r << " " << m << " ";
            report(stat);
        }

        if((m+1)%DR==0)initdata(); //initialize data every DR often

        mate();        //do a mating event
        m++;           //increment counter
    }
    cout << r << " " << m << " ";
    report(stat);
    stat.close();
    reportbest(crit);
}

crit.close();
}

int CVD(char ch){
    switch(ch){
        case 'c':
        case 'C': return(0);
            break;
        case 'g':
        case 'G': return(1);
            break;
        case 'a':
        case 'A': return(2);
            break;
        case 't':
        case 'T': return(3);
            break;
    }
    cout << "BAD DNA BASE!!! " << ((int) ch) << endl;
    return(4);
}

```



```

void readdata(){//read in the data

fstream inp;
char buf[1000];
int i,j,k,cv;

i=0;
inp.open(datafile,ios::in);
inp.getline(buf,999);
while(strlen(buf)>3){
    //cout << buf << endl;
    allclass[i]=atoi(buf);
    //cout << "Class: " << class[i] << endl;
    j=0;
    while(buf[j]!=' '){j++;}
    j++;
    allslen[i]=strlen(buf)-j;
    //cout << "Length: " << allslen[i] << endl;
    alldata[i]=new int[allslen[i]];
    for(k=0;k<allslen[i];k++)alldata[i][k]=CVD(buf[j++]);
    i++;
    inp.getline(buf,999);
}

//initialize the data sorting index
for(i=0;i<SSF;i++)DX[i]=i;

//create the current data register
for(i=0;i<SS;i++)data[i]=new int[UBSL];
}

void initialize(){//initialize full algorithm, e.g. population storage

int i;

srand48(31207813); //seed the random number generator
for(i=0;i<popsiz; i++)pop[i].create(VARS);

//create the signature buffer
sig=new double*[SS];
for(i=0;i<SS;i++)sig[i]=new double[VARS];

//create the full signature buffer
allsig=new double*[SSF];
for(i=0;i<SSF;i++)allsig[i]=new double[VARS];
}

void initpop(){//initialize the population

int i,j;

for(i=0;i<popsiz; i++){
    dx[i]=i; //make sure the sorting index is intact
    pop[i].recreate();
}
}

```

```

    fit[i]=fitness(pop[i]);
}
}

void initdata(){//make a subselection of the database

int i,j,sw,rv;

for(i=0;i<SS;i++){//pick SS data items
    rv=lrnd48()%SSF;
    sw=DX[i];DX[i]=DX[rv];DX[rv]=sw;
}

//now swap the data items into the data register
for(i=0;i<SS;i++){
    clss[i]=allclss[DX[i]];
    slen[i]=allslen[DX[i]];
    for(j=0;j<slen[i];j++)data[i][j]=alldata[DX[i]][j];
}

//and now we have a subselection of the data
}

/* This is the experimental section of the code. We run each sequence
through the FSM. It records the number of times each state is
visited. We then cluster the visitation vectors and see how well the
machine learns the correct arrangement of DNA sequences. The
clustering algorithm is K-means and the assement is the Rand index.
*/

double fitness(aut &a){//

int classn[SS];           //class number from K-means clustering
int i,j,cnt;              //loop indices
int cv[VARs];             //reporting variable
int r,s;                  //Rand index comparison variables
double rv,dv;             //return value and divisor
double ttl;               //total

    //cout << "Starting fitness evaluation" << endl;

    //gather the side effect signatures
    for(i=0;i<SS;i++){//loop over sequences
        a.reset(); //reset the automata
        //cout << i << " about to run" << endl;
        //cout << clss[i] << " " << slen[i] << endl;
        //for(j=0;j<slen[i];j++)cout << data[i][j];
        //cout << endl;
        for(j=0;j<slen[i];j++)a.run(data[i][j]); //run the sequence data
        //cout << "Ran" << endl;
        a.report(cv); //get the state-use count vector
        ttl=0.0;
        for(j=0;j<VARs;j++){
            sig[i][j]=((double)cv[j]); //store it as real data

```

```

        ttl+=sig[i][j];                //get the total length
    }
    for(j=0;j<VARS;j++)sig[i][j]/=ttl; //normalize the length
}

//perform the K-means clustering step
Kmeans(sig,SS,VARS,NSC,classn);

//compute the Rand index
cnt=0;
for(i=0;i<SS-1;i++)for(j=i+1;j<SS;j++){
    r=(classn[i]==classn[j]);
    s=(classn[i]==classn[j]);
    //cout << i/60 << " " << j/60 << " " << classn[i] << " " << classn[j];
    if(r==s)cnt++;
    //cout << " " << cnt << endl;
}

dv=((double)SS);
rv=((double)cnt);
rv=(2*rv)/dv/(dv-1);

//cout << rv << " " << endl;

return(rv);
}

double fullfitness(aut &a){//
    int classn[SSF];           //class number from K-means clustering
    int i,j,cnt;               //loop indices
    int cv[VARS];              //reporting variable
    int r,s;                   //Rand index comparison variables
    double rv,dv;              //return value and divisor
    double ttl;                //total

    //cout << "Starting fitness evaluation" << endl;

    //gather the side effect signatures
    for(i=0;i<SSF;i++){//loop over sequences
        a.reset(); //reset the automata
        for(j=0;j<allslen[i];j++)a.run(alldata[i][j]); //run the sequence data
        //cout << "Ran" << endl;
        a.report(cv); //get the state-use count vector
        ttl=0.0;
        for(j=0;j<VARS;j++){
            allsig[i][j]=((double)cv[j]); //store it as real data
            ttl+=allsig[i][j];             //get the total length
        }
        for(j=0;j<VARS;j++)allsig[i][j]/=ttl; //normalize the length
    }

    //perform the K-means clustering step
    Kmeans(allsig,SSF,VARS,NSC,classn);

```

```

//compute the Rand index
cnt=0;
for (i=0;i<SSF-1;i++)for (j=i+1;j<SSF;j++){
    r=(allclass[i]==allclass[j]);
    s=(classn[i]==classn[j]);
    //cout << i/60 << " " << j/60 << " " << classn[i] << " " << classn[j];
    if(r==s)cnt++;
    //cout << " " << cnt << endl;
}

dv=((double)SSF);
rv=((double)cnt);
rv=(2*rv)/dv/(dv-1);

//cout << rv << " " << endl;

return(rv);
}

void mate(){//perform a mating event

int i,j;           //loop indices
int sw;            //swap variable
int flag;          //sorting flag
int cp1,cp2;       //crossover points

for(i=0;i<tsize;i++){//swap a tournament to the front of the sorting order
    j=lrnd48()%popsize;
    sw=dx[i];dx[i]=dx[j];dx[j]=sw;
}

//reevaluate the fitness of the tournament members
for(i=0;i<tsize;i++){
    fit[dx[i]]=fitness(pop[dx[i]]);
}

//sort the tournament so that the two worst fitnesses are first
do {
    flag=0;
    for (i=0;i<tsize-1;i++)if (fit[dx[i]]>fit[dx[i+1]]){
        sw=dx[i];dx[i]=dx[i+1];dx[i+1]=sw;
        flag=1;
    }
}while(flag);

//perform reproduction and crossover
pop[dx[0]].copy(pop[dx[tsize-2]]);
pop[dx[1]].copy(pop[dx[tsize-1]]);
pop[dx[0]].tpc(pop[dx[1]]);

//perform mutations

```

```

    for (i=0; i<NMM; i++){
        pop[0].mutate();
        pop[1].mutate();
    }

    // assess fitness
    fit[dx[0]] = fitness(pop[dx[0]]);
    fit[dx[1]] = fitness(pop[dx[1]]);
}

void report(ostream &aus){ //interim statistical reporting

double mu, sg, bs; //mean, variance, best
int i; //loop index

mu=sg=bs=0.0;
for (i=0; i<popsiz; i++){
    mu+=fit[i];
    sg+=(fit[i]*fit[i]);
    if (fit[i]>bs) bs=fit[i];
}
mu/=((double)popsiz);
sg/=((double)popsiz);
sg-=(mu*mu);
if (sg<0) sg=0.0; else sg=sqrt(sg);
aus << mu << " " << 1.96*sg/sqrt((double)popsiz)
    << " " << sg << " " << bs << endl;
cout << mu << " " << 1.96*sg/sqrt((double)popsiz)
    << " " << sg << " " << bs << endl;
}

void reportbest(ostream &aus){ //report best machine at end

int i, bdx;
double bs;

for (i=0; i<popsiz; i++) fit[i]=fullfitness(pop[i]);

bs=0.0;
for (i=0; i<popsiz; i++) if (fit[i]>bs){
    bs=fit[i];
    bdx=i;
}

aus << bs << " fitness" << endl;
pop[bdx].write(aus);
}

```

A.9 Ignorance Side Effect Machine Evolver Code

```
/*
 * Ignorance side effect machines.
 */

//Include the relevant includes
#include <iostream>
#include <fstream>
#include <cmath>
#include <cstdlib>
#include <cstdio>
#include <ctime>
#include <cstring>

using namespace std;

#include "semI.h"
#include "Kmeans.h"

//data file to work on
#define datafile "Cprimers.dat"

//number of sequences in the data set
#define SS 200

//number of sequence classes
#define NSC 2

//population size
#define popsize 10

//tournament size
#define tsize 7

//Number of states in the finite state machine
#define VARS 12

//Number of mutations per new machine
#define NMM 1

//maximum number of mating events
#define M 20000

//reporting intervals
#define RI 100

//number of runs to perform
#define R 100

aut pop[popsize]; //the population
double fit[popsize]; //fitness array
double bestfit; //best fitness so far
```

```

int dx[popsize];           //sorting index
int *data[SS];             //DNA data
int clss[SS];              //known class data
int slen[SS];              //sequence lengths
double **sig;              //buffer for signature evaluation

void readdata();            //read the data we are training on
void initialize();          //initialize full algorithm, e.g. population storage
void initpop();             //initialize the population
double fitness(aut &a);     //compute the fitness of and automata
void mate();               //perform a mating event
void report(ostream &aus);  //interim statistical reporting
void reportbest(ostream &aus); //report best machine at end

main(){

int r,m;                   //run and mating event index variables
fstream resfyl;            //results file
double mu,sg,delta;        //statistical variables
int failures;              //number of timeouts
double N,V;                //statistical variables
char buf[60];              //string construction buffer
double p,q;                //probability of failure
fstream stat,crit;         //statistics and best machines

    readdata();             //read in the data set
    initialize();           //allocate population storage, etc.

    crit.open("best.sem",ios::out);
    for(r=0;r<R;r++){//loop over runs
        sprintf(buf,"run%d.dat",r);
        stat.open(buf,ios::out);
        initpop();
        m=0;                //start mating even counter at zero
        while(m<M){//loop until failure or success
            if(nf%RI==0){
                cout << r << " " << m << " ";
                report(stat);
            }
            mate();          //do a mating event
            m++;             //increment counter
        }
        cout << r << " " << m << " ";
        report(stat);
        stat.close();
        reportbest(crit);
    }
    crit.close();
}

int CVD(char ch){

    switch(ch){
        case 'c':

```

```

    case 'C': return(0);
        break;
    case 'g':
    case 'G': return(1);
        break;
    case 'a':
    case 'A': return(2);
        break;
    case 't':
    case 'T': return(3);
        break;
}
cout << "BAD DNA BASE!!! " << ((int) ch) << endl;
return(4);
}

void readdata(){

fstream inp;
char buf[1000];
int i,j,k,cv;

i=0;
inp.open(datafyle,ios::in);
inp.getline(buf,999);
while(strlen(buf)>3){
    //cout << buf << endl;
    class[i]=atoi(buf);
    //cout << "Class: " << class[i] << endl;
    j=0;
    while(buf[j]!=' '){j++;}
    j++;
    slen[i]=strlen(buf)-j;
    //cout << "Length: " << slen[i] << endl;
    data[i]=new int[slen[i]];
    for(k=0;k<slen[i];k++)data[i][k]=CVD(buf[j++]);
    i++;
    inp.getline(buf,999);
}

}

void initialize(){//initialize full algorithm, e.g. population storage

int i;

srand48(31207813); //seed the random number generator
for(i=0;i<popsize;i++)pop[i].create(VARS);

//create the signature buffer
sig=new double*[SS];
for(i=0;i<SS;i++)sig[i]=new double[VARS];
}

```



```

void initpop(){//initialize the population

int i,j;

    for(i=0;i<popsiz; i++){
        dx[i]=i; //make sure the sorting index is intact
        pop[i].recreate();
        fit[i]=fitness(pop[i]);
    }
}

/* This is the experimental section of the code. We run each sequence
through the FSM. It records the number of times each state is
visited. We then cluster the visitation vectors and see how well the
machine learns the correct arrangement of DNA sequences. The
clustering algorithm is K-means and the assement is the Rand index.
*/

double fitness(auto &a){//

int classn[SS];           //class number from K-means clustering
int i,j,cnt;              //loop indices
int cv[VARs];             //reporting variable
int r,s;                  //Rand index comparison variables
double rv,dv;             //return value and divisor
double ttl;               //total

    //cout << "Starting fitness evaluation" << endl;

    //gather the side effect signatures
    for(i=0;i<SS;i++){//loop over sequences
        a.reset(); //reset the automata
        for(j=0;j<slen[i];j++)a.run(data[i][j]); //run the sequence data
        a.report(cv); //get the state-use count vector
        ttl=0.0;
        for(j=0;j<VARs;j++){
            sig[i][j]=((double)cv[j]); //store it as real data
            ttl+=sig[i][j]; //get the total length
        }
        for(j=0;j<VARs;j++)sig[i][j]/=ttl; //normalize the length
    }

    //perform the K-means clustering step
    Kmeans(sig,SS,VARs,NSC,classn);

    //compute the Rand index
    cnt=0;
    for(i=0;i<SS-1;i++)for(j=i+1;j<SS;j++){
        r=(classn[i]==classn[j]);
        s=(classn[i]==classn[j]);
        //cout << i/60 << " " << j/60 << " " << classn[i] << " " << classn[j];
        if(r==s)cnt++;
        //cout << " " << cnt << endl;
    }
}

```

```

    dv=((double)SS);
    rv=((double)cnt);
    rv=(2*rv)/dv/(dv-1);

    //cout << rv << " " << endl;

    return(rv);
}

void mate(){//perform a mating event

int i,j;           //loop indices
int sw;            //swap variable
int flag;          //sorting flag
int cp1,cp2;       //crossover points

    for(i=0;i<tsize;i++){//swap a tournament to the front of the sorting order
        j=lrand48()%popsize;
        sw=dx[i];dx[i]=dx[j];dx[j]=sw;
    }

    //reevaluate the fitness of the tournament members
    for(i=0;i<tsize;i++){
        fit[dx[i]]=fitness(pop[dx[i]]);
    }

    //sort the tournament so that the two worst fitnesses are first
    do {
        flag=0;
        for(i=0;i<tsize-1;i++)if(fit[dx[i]]>fit[dx[i+1]]){
            sw=dx[i];dx[i]=dx[i+1];dx[i+1]=sw;
            flag=1;
        }
    }while(flag);

    //perform reproduction and crossover
    pop[dx[0]].copy(pop[dx[tsize-2]]);
    pop[dx[1]].copy(pop[dx[tsize-1]]);
    pop[dx[0]].tpc(pop[dx[1]]);

    //perform mutations
    for(i=0;i<MM;i++){
        pop[0].mutate();
        pop[1].mutate();
    }

    //assess fitness
    fit[dx[0]]=fitness(pop[dx[0]]);
    fit[dx[1]]=fitness(pop[dx[1]]);
}

```

```
void report(ostream &aus){//interim statistical reporting
```

```
double mu,sg,bs; //mean, variance, best
int i; //loop index
```

```
mu=sg=bs=0.0;
for(i=0;i<popsiz; i++){
    mu+=fit[i];
    sg+=(fit[i]*fit[i]);
    if(fit[i]>bs)bs=fit[i];
}
mu/=((double)popsiz);
sg/=((double)popsiz);
sg-=(mu*mu);
if(sg<0)sg=0.0; else sg=sqrt(sg);
aus << mu << " " << 1.96*sg/sqrt((double)popsiz)
    << " " << sg << " " << bs << endl;
cout << mu << " " << 1.96*sg/sqrt((double)popsiz)
    << " " << sg << " " << bs << endl;
}
```

```
void reportbest(ostream &aus){//report best machine at end
```

```
int i,bdx;
double bs;
```

```
bs=0.0;
for(i=0;i<popsiz; i++){
    bs=fit[i];
    bdx=i;
}
aus << bs << " fitness" << endl;
pop[bdx].write(aus);
}
```

A.10 Ignorance Side Effect Machine Population Creator Code

```
/* Side effect machine class of finite state automata
 *
 * code file
 */

#include <iostream>
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <cctype>
#include <cmath>

#include "semI.h"

//constructors and destructors
aut::aut(){//create an empty automata

    blank();
}

aut::aut(int nv){//create an automata with nv states

    n=0;
    create(nv);
}

aut::aut(const aut &other){//copy constructor

    int i,j;

    n=other.n;
    cur=other.cur;
    cntr=new int[n];
    ig=new int[n];
    states=new int*[n];
    for(i=0;i<n;i++){
        states[i]=new int[TR];
        for(j=0;j<TR;j++)states[i][j]=other.states[i][j];
        cntr[i]=other.cntr[i];
        ig[i]=other.ig[i];
    }
}

aut::~aut(){//deallocate the automata

    clear();
}

//management routines
```

```

void aut::blank(){//create an empty automata

    n=0;
}

void aut::clear(){//clear the automata

    if(n){
        for(int i=0;i<n;i++)delete [] states[i];
        delete [] states;
        delete [] cntr;
        delete [] ig;
    }
    n=0;
}

void aut::create(int nv){//create an automata with nv states

    if(n)clear();
    n=nv;
    states=new int*[n];
    for(int i=0;i<n;i++)states[i]=new int[TR];
    cntr=new int[n];
    ig=new int[n];
    recreate();
}

void aut::recreate(){//randomize an allocated automata

    int i,j;

    if(n){//failsafe the recreation of empty automata
        for(i=0;i<n;i++){//loop over states
            ig[i]=lrand48()%2; //create ignorage
            for(j=0;j<TR;j++)states[i][j]=lrand48()%n; //make transitions
        }
    }
}

void aut::copy(aut &other){//make a copy of another automata

    int i,j;

    if(n!=other.n)create(other.n);
    cur=other.cur;
    for(i=0;i<n;i++){
        for(j=0;j<TR;j++)states[i][j]=other.states[i][j];
        cntr[i]=other.cntr[i];
        ig[i]=other.ig[i];
    }
}

//use routines
void aut::reset(){//set state to zero and clear counters

```

```

    cur=0; //set current state to zero
    for(int i=0;i<n;i++)cntr[i]=0; //zero the counters
    cntr[0]=1; //record the starting state
}

void aut::run(int v){//run on input value v

    cur=states[cur][v]; //make the state transition
    cntr[cur]++; //record the entry into the state
}

void aut::report(int *rp){//report the state useage vectors

    //not that rp is presumed to be long enough
    for(int i=0;i<n;i++)rp[i]=cntr[i]*ig[i];
}

//genetic operations
void aut::tpc(aut &other){//perform two point crossover

int i,j,sw,cp1,cp2;

    cp1=rand48()%n; //find first crossover point
    cp2=rand48()%n; //find second crossover point
    if(cp1>cp2){sw=cp1;cp1=cp2;cp2=sw;} //order them
    for(i=cp1;i<=cp2;i++){
        //crossover the ignorage
        sw=ig[i];ig[i]=other.ig[i];other.ig[i]=sw;
        for(j=0;j<TR;j++){//cross over the transitions
            sw=states[i][j];
            states[i][j]=other.states[i][j];
            other.states[i][j]=sw;
        }
    }
}

void aut::mutate(){//perform a transition and ignorage mutation

int who;

    who=rand48()%n; //pick state
    states[who][rand48()%TR]=rand48()%n; //one new transition
    ig[who]=rand48()%2; //50% change ignorage
}

//input and output
void aut::write(ostream &aus){//output method

int i,j;

    aus << n << " states" << endl;
    for(i=0;i<n;i++){
        aus << i << ":";

```

```

        aus << ig[i];
        for(j=0;j<TR;j++)aus << "," << states[i][j];
        aus << endl;
    }
}

```

```

void aut::read(istream &inp){//input method

```

```

    char buf[1000];
    int i,j,k;

    inp.getline(buf,999);
    create(atoi(buf));
    for(i=0;i<n;i++){
        inp.getline(buf,999);
        j=0;
        while(buf[j]!=':')j++;
        j++;
        ig[i]=atoi(buf+j);
        while(isdigit(buf[j]))j++;
        for(k=0;k<TR;k++){
            j++;
            states[i][k]=atoi(buf+j);
            while(isdigit(buf[j]))j++;
        }
    }
}

```

A.11 Ignorance Side Effect Machine Population Creator Header File

```
/* Side effect machine class of finite state automata
 * header file
 */
#ifndef SEM_H
#define SEM_H

using namespace std;

//number of transitions out per state
#define TR 4

class aut { //side effect machine class

public:

    //constructors and destructors
    aut(); //create an empty automata
    aut(int nv); //create an automata with nv states
    aut(const aut &); //copy constructor
    ~aut(); //deallocate the automata

    //management routines
    void blank(); //create an empty automata
    void clear(); //clear the automata
    void create(int nv); //create an automata with nv states
    void recreate(); //randomize an allocated automata
    void copy(aut &other); //make a copy of another automata

    //use routines
    void reset(); //set state to zero and clear counters
    void run(int v); //run on input value v
    void report(int *rp); //report the state usage vectors

    //genetic operations
    void tpc(aut &other); //perform two point crossover
    void mutate(); //perform a transition mutation

    //input and output
    void write(ostream &aus);
    void read(istream &inp);

private:

    int n; //number of states
    int cur; //current state
    int **states; //state vector
    int *ig; //ignorance vector
    int *cntr; //counter vector
};
#endif /* SEM_H */
```