1

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services Branch

Direction des acquisitions et
des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file: Votre référence

Our file: Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

PROGRAM INDUCTION USING AN UNMODIFIED GENETIC ALGORITHM:
A RATIONAL RECONSTRUCTION OF GENETIC PROGRAMMING THROUGH
THE USE OF A STATIC PARSE TREE STRUCTURE

By

MARK WINEBERG, B.Sc.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

August 11, 1993

© copyright

1993, Mark Wineberg

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

THE AUTHOR HAS GRANTED AN
IRREVOCABLE NON-EXCLUSIVE
LICENCE ALLOWING THE NATIONAL
LIBRARY OF CANADA TO
REPRODUCE, LOAN, DISTRIBUTE OR
SELL COPIES OF HIS/HER THESIS BY
ANY MEANS AND IN ANY FORM OR
FORMAT, MAKING THIS THESIS
AVAILABLE TO INTERESTED
PERSONS.

L'AUTEUR A ACCORDE UNE LICENCE
IRREVOCABLE ET NON EXCLUSIVE
PERMETTANT A LA BIBLIOTHEQUE
NATIONALE DU CANADA DE
REPRODUIRE, PRETER, DISTRIBUER
OU VENDRE DES COPIES DE SA
THESE DE QUELQUE MANIERE ET
SOUS QUELQUE FORME QUE CE SOIT
POUR METTRE DES EXEMPLAIRES DE
CETTE THESE A LA DISPOSITION DES
PERSONNE INTERESSEES.

THE AUTHOR RETAINS OWNERSHIP
OF THE COPYRIGHT IN HIS/HER
THESIS. NEITHER THE THESIS NOR
SUBSTANTIAL EXTRACTS FROM IT
MAY BE PRINTED OR OTHERWISE
REPRODUCED WITHOUT HIS/HER
PERMISSION.

L'AUTEUR CONSERVE LA PROPRIETE
DU DROIT D'AUTEUR QUI PROTEGE
SA THESE. NI LA THESE NI DES
EXTRAITS SUBSTANTIELS DE CELLE-
CI NE DOIVENT ETRE IMPRIMES OU
AUTREMENT REPRODUITS SANS SON
AUTORISATION.

ISBN 0-315-98562-3

Canada

Name Mark Lorne Winstony - Program Evaluation US-13-99 Unmodified Grade 12, 1999

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Applied Sciences: Computer Science

SUBJECT TERM

0984

SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language
 General 0679
 Ancient 0289
 Linguistics 0290
 Modern 0291
Literature
 General 0401
 Classical 0294
 Comparative 0295
 Medieval 0297
 Modern 0298
 African 0316
 American 0591
 Asian 0305
 Canadian (English) 0352
 Canadian (French) 0355
 English 0593
 Germanic 0311
 Latin American 0312
 Middle Eastern 0315
 Romance 0313
 Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion
 General 0318
 Biblical Studies 0321
 Clergy 0319
 History of 0320
 Philosophy of 0322
Theology 0469

SOCIAL SCIENCES

American Studies 0323
Anthropology
 Archaeology 0324
 Cultural 0326
 Physical 0327
Business Administration
 General 0310
 Accounting 0272
 Banking 0770
 Management 0454
 Marketing 0338
Canadian Studies 0385
Economics
 General 0501
 Agricultural 0503
 Commerce-Business 0505
 Finance 0508
 History 0509
 Labor 0510
 Theory 0511
Folklore 0358
Geography 0366
Gerontology 0351
History
 General 0578

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science
 General 0615
 International Law and Relations 0616
 Public Administration 0617
Recreation 0814
Social Work 0452
Sociology
 General 0626
 Criminology and Penology 0627
 Demography 0938
 Ethnic and Racial Studies 0631
 Individual and Family Studies 0628
 Industrial and Labor Relations 0629
 Public and Social Welfare 0630
 Social Structure and Development 0700
 Theory and Methods 0344
Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture
 General 0473
 Agronomy 0285
 Animal Culture and Nutrition 0475
 Animal Pathology 0476
 Food Science and Technology 0359
 Forestry and Wildlife 0478
 Plant Culture 0479
 Plant Pathology 0480
 Plant Physiology 0817
 Range Management 0777
 Wood Technology 0746
Biology
 General 0306
 Ancient 0287
 Biostatistics 0308
 Botany 0309
 Cell 0379
 Ecology 0329
 Entomology 0353
 Genetics 0369
 Limnology 0793
 Microbiology 0410
 Molecular 0307
 Neuroscience 0317
 Oceanography 0416
 Physiology 0433
 Radiation 0821
 Veterinary Science 0778
 Zoology 0472
Biophysics
 General 0786
 Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Polynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences
 General 0566
 Audiology 0300
 Chemotherapy 0992
 Dentistry 0567
 Education 0350
 Hospital Management 0769
 Human Development 0758
 Immunology 0982
 Medicine and Surgery 0564
 Mental Health 0347
 Nursing 0569
 Nutrition 0570
 Obstetrics and Gynecology 0380
 Occupational Health and Safety 0354
 Ophthalmology 0381
 Pathology 0571
 Pharmacology 0419
 Pharmacy 0572
 Physical Therapy 0382
 Public Health 0573
 Radiology 0574
 Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences
 Chemistry
 General 0485
 Agricultural 0749
 Analytical 0486
 Biochemistry 0487
 Inorganic 0488
 Nuclear 0738
 Organic 0490
 Pharmaceutical 0491
 Physical 0494
 Polymer 0495
 Radiation 0754
 Mathematics 0405
 Physics
 General 0605
 Acoustics 0986
 Astronomy and Astrophysics 0606
 Atmospheric Science 0608
 Atomic 0748
 Electronics and Electricity 0607
 Elementary Particles and High Energy 0798
 Fluid and Plasma 0759
 Molecular 0609
 Nuclear 0610
 Optics 0752
 Radiation 0756
 Solid State 0611
 Statistics 0463
Applied Sciences
 Applied Mechanics 0346
 Computer Science 0984

Engineering
 General 0537
 Aerospace 0538
 Agricultural 0539
 Automotive 0540
 Biomedical 0541
 Chemical 0542
 Civil 0543
 Electronics and Electrical 0544
 Heat and Thermodynamics 0348
 Hydraulic 0545
 Industrial 0546
 Marine 0547
 Materials Science 0794
 Mechanical 0548
 Metallurgy 0743
 Mining 0551
 Nuclear 0552
 Packaging 0549
 Petroleum 0765
 Sanitary and Municipal 0554
 System Science 0790
Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

The undersigned hereby recommend to
The faculty of Graduate Studies and Research
acceptance of the thesis,

**Program Induction Using An Unmodified Genetic Algorithm:
A Rational Reconstruction of Genetic Programming Through the use of a Static
Parse Tree Structure**

submitted by
Mark Wineberg
in partial fulfilment of the requirements
for the degree of Master of Computer Science

A handwritten signature in black ink, appearing to read "J. R. Pung", written over a horizontal line.

Director, School of Computer Science

A handwritten signature in black ink, appearing to read "Frank Offord", written over a horizontal line.

Thesis Supervisor

Carleton University

August 11, 1993

There have been many attempts to implement the task of program induction using a Genetic Algorithm (GA). The most recent attempt is Genetic Programming (GP). GP uses a modification of the GA to breed successive populations of (initially arbitrary) programs until a program that solves the problem emerges. GP, however, does not follow the GA model exactly. It has a very different "crossover" and uses a variable structured, non-linear chromosome. Consequently, the behavior of GP is not well understood, and the theoretical underpinnings of the GA model cannot be directly applied to GP. It would be interesting to understand which of the innovations utilized in GP are required to evolve a population of programs towards the correct general solution of the task given. To achieve this, a technique is developed that encodes programs, which are used by GP as chromosomes, into linear chromosomes with fixed internal structures. A canonical GA is then applied to the new chromosomes, and a test suite of program induction problems are solved. The success of the implementation shows that the special crossover used in GP is not necessary to solve program induction through a GA. Finally, Introns, and the Transposition Operator, two new tools that may be used to improve the performance of some GA applications, were introduced.

ACKNOWLEDGEMENTS

iv

I greatly acknowledge the support and insights provided by my supervisor, Franz Oppacher, who, with simple questions, planted the ideas which eventually blossomed into the main concept behind the thesis. I also wish to thank Una-May O'Reilly, who's presentation, and discussions afterward, got me thinking about Genetic Programming, and for her friendship, and support throughout the process. Finally I must thank my parents, Sam, and Sylvia Wineberg; without their financial aid, and constant prodding, moral support, and overall assistance, this thesis would not have been possible.

TABLE OF CONTENTS

	PAGE
ACCEPTANCE SHEET	ii
ABSTRACT	iii
ACKNOWLEDGEMENT	iv
TABLE OF CONTENTS	v
LIST OF TABLES	ix
LIST OF FIGURES	x
 CHAPTER I: INTRODUCTION	 1
 CHAPTER II: GENETIC ALGORITHMS	
2.1 INTRODUCTION TO GENETIC ALGORITHMS	
2.1.1 OPTIMIZATION PROBLEMS AND SEARCH	
HEURISTICS	3
2.1.2 AN ANALYSIS OF THE MAJOR SEARCH HEURISTICS	
FOR OPTIMIZATION PROBLEMS	5
2.1.3 RELAXING THE CRITERIA:	
A SOLUTION THAT IS GOOD ENOUGH	7
2.1.4 DARWINIAN EVOLUTION	7
2.2 THE GENETIC ALGORITHM	
2.2.1 OVERVIEW	9
2.2.2 BIOLOGICAL TERMINOLOGY	10
2.2.3 ENCODING THE CHROMOSOME	12
2.2.4 SELECTION METHODS FOR REPRODUCTION	13
2.2.5 THE FITNESS FUNCTION	20
2.2.6 CROSSOVER AND REPRODUCTION	22
2.2.7 MUTATION	24
2.2.8 CONVERGENCE, AND A POPULATION OF	
SOLUTIONS	25
2.2.9 TERMINATION CONDITIONS	27
2.2.10 PUTTING IT TOGETHER: THE GENETIC	
ALGORITHM	27
2.3 GENETIC ALGORITHM THEORY	30
 CHAPTER III: A BRIEF REVIEW OF PROGRAM EVOLUTION	
3.1 INTRODUCTION	36
3.2 CONTEXT SENSITIVE INTERPRETATION AND ORDER	
DEPENDENCY	36
3.3 PROGRAM EVOLUTION SYSTEMS	37
3.3.1 JB AND TB	37
3.3.2 A GA THAT MANIPULATES LISP CODE	39
3.3.3 GENETIC PROGRAMMING	39
3.4 SUMMARY	41

	PAGE
CHAPTER IV: GENETIC PROGRAMMING	
4.1 INTRODUCTION TO GENETIC PROGRAMMING.....	42
4.2 THE USE OF LISP S-EXPRESSIONS AND PARSE TREES	
4.2.1 PROGRAMS AS CHROMOSOMES.....	43
4.2.2 PARSE TREES AS CHROMOSOMES	43
4.2.3 LISP PROGRAMS	46
4.3 THE GENETIC PROGRAMMING ALGORITHM	
4.3.1 OVERVIEW OF THE ALGORITHM	48
4.3.2 THE OPERATOR AND TERMINAL SETS	49
4.3.3 RANDOM GENERATION OF THE POPULATION OF LISP STATEMENTS	51
4.3.4 REPRODUCTION.....	52
4.3.5 THE FITNESS FUNCTION	54
4.4 THE DIFFERENCES BETWEEN GENETIC PROGRAMMING AND GENETIC ALGORITHMS	55
4.5 NICHES AND PROBABILITY OF CONVERGENCE	57
4.6 APPLICATIONS USED IN THE THESIS	60
4.7 A NEW SCHEMA DEFINITION: AN ATTEMPT TO EXPLAIN GP USING THE SCHEMA THEOREM	64
CHAPTER V: FIXED LENGTH CHROMOSOMAL GENETIC PROGRAMMING	
5.1 KEEPING CLOSE TO GA: AN EP-I OUTLINE	66
5.2 A SIMPLE DESCRIPTION OF EP-I	
5.2.1 THE FIXED HEIGHT EXPANDED PARSE TREE	68
5.2.2 THE SELECTOR OPERATOR: EMBEDDING THE PARSE TREE	69
5.2.3 THE COMPATIBILITY OF THE EXPANDED PARSE TREE WITH TRADITIONAL GENETIC OPERATORS	73
5.3 SYMBOLIC ALPHABETS AND DECODING	
5.3.1 SYMBOLIC ALPHABETS	74
5.3.2 THE ENCODING SYSTEM FOR EP-I	76
5.3.3 THE DECODER	77
5.4 N-ARY PARSE TREES	
5.4.1 ENCODING A FULL N-ARY PARSE TREE	81
5.4.2 THE SELECTOR OPERATORS IN A FULL N-ARY PARSE TREE	85
5.4.3 COMPARISON OF THE THREE ENCODING METHODS	91
5.4.4 SIZE OF THE SEARCH SPACE: THE EFFECTS OF A SYMBOLIC GENE	92

	PAGE
5.5 THE BEHAVIOR OF GENETIC OPERATORS IN EP-I	
5.5.1 SYNTACTIC CLOSURE REVISITED	94
5.5.2 MUTATION IN EP-I	94
5.5.3 CROSSOVER IN EP-I	98
5.5.4 THE COUPLED 2-POINT CROSSOVER	102
CHAPTER VI: NOVEL BIOLOGICAL ANALOGIES AND EXTENSIONS TO EP-I	
6.1 INTRODUCTION	105
6.2 INTRONS AND EXONS	
6.2.1 THE DEFINITIONS OF INTRONS AND EXONS..	105
6.2.2 INTRONS IN EP-I	107
6.2.3 THE EFFECTS OF INTRONS ON EP-I	108
6.2.4 INTRONS AND THE MUTATION OPERATOR	110
6.2.5 INTRONS AND THE CROSSOVER OPERATOR	113
6.3 INVERSION AND GENE JUMPING	
6.3.1 INVERSION	117
6.3.2 A MISLABELED GENETIC OPERATOR	120
6.3.3 TRANSPOSITION, A.K.A. JUMPING GENES	122
6.3.4 THE TRANSPOSITION OPERATOR: AN EXTENSION TO EP-I	124
6.4 THE EP-I SIMULATION OF GP	
6.4.1 THE GP-CROSSOVER	128
6.4.2 A TIME COMPLEXITY ANALYSIS OF THE GP SIMULATION	131
6.4.3 SIMULATING GP	132
CHAPTER VII: EXPERIMENTATION	
7.1 INTRODUCTION	138
7.2 DESIGN ELEMENTS	
7.2.1 SUMMARY OF THE TEST SUITES	139
7.2.2 SYSTEM DESIGN OPTIONS	142
7.3 THE EXPERIMENTS	143
7.4 RESULTS	
7.4.1 A DESCRIPTION OF A SUCCESSFUL RUN	147
7.4.2 A SUMMARY OF THE PROBABILITY OF CONVERGENCE FOR THE EXPERIMENTS	151
CHAPTER XIII: DISCUSSIONS AND FUTURE WORK	
8.1 THE PROPERTIES OF EP-I AND THEIR IMPLICATIONS	
8.1.1 AN UNMODIFIED GA CAN PERFORM PROGRAM INDUCTION	153
8.1.2 EP-I IS NOT A RANDOM SEARCH	155
8.1.3 SUNDRY OBSERVATIONS ON THE EXPERIMENTS AND ANECDOTES	156

	PAGE
8.2 COMPARING GP WITH EP-I	
8.2.1 GENERAL DIFFERENCES.....	158
8.2.2 THE SCHEMATA USED BY EP-I.....	161
8.2.3 ARE SUBTREES NATURAL BUILDING BLOCKS FOR PROGRAM INDUCTION?.....	162
8.2.4 ANALYZING GP THROUGH EP-I SIMULATION	166
8.3 CONCLUSION AND FUTURE WORK	
8.3.1 CONCLUSION.....	168
8.3.2 FUTURE WORK.....	170
REFERENCES.....	171

LIST OF TABLES

TABLE	DESCRIPTION	PAGE
Table 2.1	Examples of Problems From Artificial Intelligence that can be Turned Into An Optimization Problem	4
Table 2.2	User Defined Parameters	29
Table 2.3	Design Options	30
Table 7.1	Parameter Settings.....	146
Table 7.2	Average Run for 6 Bit Multiplexor Problem	149
Table 7.3	Summary of Results	152
Table 8.1	Differences between GA and GP	158
Table 8.2	The Maximum and Average Adjusted Fitness for Selected Generations from Experiment 4-a4	160

LIST OF FIGURES

FIGURE	DESCRIPTION	PAGE
Figure 2.1	Chromosome Features.....	11
Figure 2.2	The Effects of Fitness Proportional Selection.....	15
Figure 2.3	Two Examples of the Effects of Rank Selection	17
Figure 2.4	1-Point Crossover.....	23
Figure 2.5	Uniform Crossover With a 0.2 Probability for Gene Exchange	24
Figure 2.6	An Expansion of Step 4a Using 1-Point Crossover	28
Figure 4.1	A Recursive Algorithm for Collapsing a Parse Tree	44
Figure 4.2	GP Crossover Example for Parse Trees	45
Figure 4.3	Components of LISP S-Expressions	47
Figure 4.4	A LISP Macro That Reverse the Evaluation Order of its Arguments	48
Figure 4.5	An Example of GP Schemata.....	64
Figure 5.1	A Parse Tree Embedded in an Expanded Parse Tree	72
Figure 5.2	A Template for Finding the Position of Children in the Chromosome	79
Figure 5.3	A Breadth-First Encoding of a Parse Tree in a Chromosome.....	80
Figure 5.4	Using A LISP Macro To Ignore the 3rd Argument	83
Figure 5.5	Decoding a Chromosome with Parameter Location Vectors	85
Figure 5.6	Decoding a Chromosome Using Selection Method 2	87
Figure 5.7	Decoding a Chromosome Using Selection Method 3	89
Figure 5.8	The Effect of Mutation on Decoding	96
Figure 5.9	An Example of Crossover	99
Figure 5.10	Subtrees Which are Crossed Over, are Embedded at Contiguous Loci	103
Figure 6.1	Example Chromosome with Introns and Exons Detailed	108
Figure 6.2	The Effects of Mutation on Introns and Exons	111
Figure 6.3	Crossover, When Both Crossover Points are Introns, Can Produce New Children	115
Figure 6.4	Crossover When Both Crossover Points are Tail Introns: No Effect on Phenotype	116
Figure 6.5	An Example of Inversion, as used in a Genetic Algorithm	117
Figure 6.6	Inversion as a Crossover Suppressor	119

LIST OF FIGURES

xi

FIGURE	DESCRIPTION	PAGE
Figure 6.7	Invisible and Regular Transposition Between Siblings	126
Figure 6.8	Transposition Within Level	127
Figure 6.9	An Example of GP crossover simulation in EP-I	133
Figure 8.1	The Building Blocks Favored by 1-Point Crossover: Crossing Subtree Boundaries	164
Figure 8.2	The Higher The Probability of Gene Exchange in Uniform Crossover, The Higher The Probability That Subtrees Will Be Disrupted	165

CHAPTER I: INTRODUCTION

Since the mid-Nineteen Eighties the Genetic Algorithm has grown to become a successful search technique with applications in a wide variety of fields. GA has been applied to biology, engineering and operations research, image processing and pattern recognition, the physical sciences and the social sciences ([Goldberg 89] pp. 126 - 127). The **Genetic Algorithm** (which shall often be abbreviated to **GA** in this paper) is a probabilistic heuristic, that searches a very large (but finite) search space, for maximal solutions to optimization problems. The GA method is inspired by the evolutionary process of Darwinian natural selection, and neo-Darwinian genetics.

It is enticing to use the evolutionary process, through the GA search technique, to evolve a set of randomly created programs towards solving a given task. The process by which a general computer program takes a specific task as input, but is not given the method to solve the task, and creates another computer program that does solve it, is known as **program induction**. Using evolution, to solve program induction, is an obviously desirable goal, and one envisioned by researchers in the field in the sixties [Fogel, et. al. 66]. Recently, methods for the evolution of computer programs have been developed that show much promise. The field is still in its infancy. However, many non-trivial problems taken from many areas of machine learning have been solved by applying evolution to computer programs.

The most successful method developed so far is '**Genetic Programming**' (which shall often be abbreviated to **GP** in this paper), developed by John R. Koza in 1989. Genetic Programming has been successfully applied to the problems of: optimal control, planning sequence induction, symbolic regression, automatic programming, game playing strategy, symbolic integration and differentiation, inverse problems, discovering

mathematical identities, classification and decision tree induction, evolution of emergent behavior, and automatic programming of cellular automata ([Koza 92] pg. 15). However, GP, while rooted in GA techniques, deviates from it enough, that the theoretical understanding developed for GA can not be readily applied. The success of GP is a result of a purely empirical study, and no true understanding of why it works exists.

To help develop a better understanding of the evolution of computer programs, through natural selection, the methods used in successful algorithms, such as Genetic Programming, should be identified, and evaluated, for their effectiveness, or necessity, in solving program induction. The differences between GP, and a canonical GA, should be studied, to see which of the new, or modified, techniques, introduced by GP, are fundamental to the process of program induction through evolution, and which are just expedient methods, used to allow the system to function, and are thus replaceable.

To accomplish this, a rational reconstruction of Koza's Genetic Programming algorithm is produced, staying as close as possible to the canonical GA. The resulting system is called **EP-I** which stands for '**Evolutionary Programming with Introns**'. EP-I performs program induction using only the genetic operators and selection methods that have been used in the regular, and well studied, GA systems. Therefore EP-I inherits all of the properties of, as well as the theoretical results produced for, GA. By examining the behavior of EP-I, it should be possible to determine whether GP can be reduced to an instantiation of the canonical GA, for solving program induction through evolution, or, if not, which features, not shared by GP with GA, are required. Also, by providing an alternative viewpoint, further insight should be gained, into both the behavior of GP, and, hopefully, into the nature of the GA itself.

CHAPTER II: GENETIC ALGORITHMS

2.1 INTRODUCTION TO GENETIC ALGORITHMS

2.1.1 OPTIMIZATION PROBLEMS AND SEARCH HEURISTICS

The GA is an example of a search heuristic used to solve an optimization problem. In an optimization problem, there are many possible solutions, and the goal is to find the best of those solutions. Solving an optimization problem is analogous to (and can be seen as) finding the maximum (or minimum) of a function: each set of input values, to the function, is a possible solution to the optimization problem, and the function determines the quality of each potential solution. The set of input values that produce a maximum (or minimum) function value, is the solution that solves the optimization problem. Many problems in Computer Science, especially in Artificial Intelligence, are optimization problems or can easily be turned into an optimization problem. Some examples are given in Table 2.1.

Search heuristics are the mainstay of Artificial Intelligence. A search heuristic is a 'rule of thumb' used to reduce the size of the search space. They are different from data base searching techniques. Data bases rely on a linear sized search space, so a data base can, in the worst case, search through all the elements. A search heuristic is used when the search space is too large to examine every element. Different search heuristics are used, depending on the nature of the data being searched; the organization, or lack of organization, of the search space; and the quantity of information, about any implicit order

in the system, which can be gleaned from the elements. Binary search is a simple example of a search heuristic.

Table 2.1 Examples of Problems that can be Turned Into An Optimization Problem

Problem		Description: As An Optimization Problem
Bin Packing	Pot. Sol.	The order that the boxes are inserted in the bin
	Max. F^n	The number of boxes placed in the bin after insertion
Graph Colouring	Pot. Sol.	A colouring of the map
	Min. F^n	The number of contiguous spaces on the map. which use the same colour
The Traveling Salesman Problem	Pot. Sol.	Each tour of all the cities
	Min. F^n	The time that the tour takes
Clustering Points in a Graph	Pot. Sol.	Each possible division of the set of points
	Min. F^n	Both, the number of sets, and, the distances between the points in a set
Gaming Strategies	Pot. Sol.	The strategy used to play a game
	Max. F^n	The number of games won, or the number of points accrued

Legend: Pot. Sol. Potential Solution
 Max. F^n Maximization Function
 Min. F^n Minimization Function

2.1.2 AN ANALYSIS OF THE MAJOR SEARCH HEURISTICS FOR OPTIMIZATION PROBLEMS

According to Goldberg ([Goldberg 89] p.2) there are three types of search techniques that work on an optimization problem: calculus methods, enumerative methods, and random methods.

There are two basic calculus methods, the analytical method (which Goldberg calls the indirect method) and the iterative method (which Goldberg calls the direct method).

The analytical method first finds extrema points by deriving a set of equations that measure the gradient of the function, setting each equation equal to zero (which restricts the equation to the points that have a level slope, which are either extrema, or plateaus), and then solving the set of equations to determine the extrema points. Then, the maxima points are selected from the extrema points by taking the second derivative of the function, evaluating it at the extrema points, and any point which produces a negative result is a maximum. Finally, the largest of this set of points will be the maximum.

The iterative approach selects a sequence of potential solutions and uses the gradient of the function to pick the next point to test. This is a form of hill-climbing. The points around the current best guess are sampled, the direction with the greatest slope is determined (the sampled point that has the greatest value, will have the greatest slope) and the algorithm makes a new guess in the direction of the greatest slope. This continues until all points are below the current guess, at which point, the hill has been climbed, and the maximum has been found.

Both of these techniques are limited to the domain of differentiable functions. Many problems have discontinuities, severe noisiness, and discrete domains and ranges (for example the bin packing problem and the traveling salesman problem mentioned above) and thus are not differentiable. In addition the analytical method is only useful for a

small subset of all possible differentiable functions, those whose gradients when set to zero can be solved analytically; and the iterative method can easily get stuck at local maxima, and miss the global maximum.

The enumerative method is the simplest of the three and has already been mentioned; the solution space is examined one element at a time, until the entire space is searched (or until the target is found). These include the many search methods developed for searching hierarchical structures, such as branch-and-bound, and mini-max. However, as mentioned before, the problems encountered in Artificial Intelligence are usually exponential in size, leading to huge search spaces, which are not feasible to fully explore in a reasonable amount of time.

The random method selects a fixed number of random points in the search space. Each point is examined, and the point that produces the best solution, is chosen. This method is ultimately defeated by the size of the search space, just as the enumerative method was. The chance of choosing a sample set, which holds a member that is large enough to be considered a good answer, is small, unless a very large sample set is chosen, which, again, is too costly in time.

The Genetic Algorithm is a directed probabilistic search technique. This is not the same as the random method, because it uses information, extracted from the evaluation of the solution points, to direct the algorithm to decide which new points to select. In this way the algorithm does an intelligent, not a random, search of the solution space. It does not look at every point, as do the enumerative methods¹, and, unlike the calculus methods,

¹ Some of the enumerative methods developed for hierarchical structures, such as branch-and-bound, are also directed search heuristics (although not probabilistic heuristics). While the heuristic can result in a search of the entire space, it tries to restrict itself to searching likely areas, where the solution may be found. Therefore, it is unfair to claim that they need to search through all points of the search space. However, these search heuristics are only valid for data organized in tree structures, and so are limited in their application. In particular, program induction of LISP expressions cannot be searched

it is valid over a discontinuous, and non-differentiable, domain, and range. It is also resistant to being trapped in local maxima, unlike the hill climbing methods. The Genetic Algorithm is, therefore, a very robust technique, applicable over a very wide variety of applications.

2.1.3 RELAXING THE CRITERIA: A SOLUTION THAT IS GOOD ENOUGH

The Genetic Algorithm achieves its robustness at a cost: it may not find the optimal solution. This is not as bad as it sounds. Depending on the search space, the algorithm may indeed find the optimal solution, but, in many cases, a solution that is "good enough", is satisfactory. We have already seen a search technique that does not find the optimal solution: the random search heuristic. Often an optimization problem needs an answer that is proficient, not perfect. This is frequently the case in real world problems seen in Engineering and many branches of Artificial Intelligence, such as Machine Learning, gaming, and robot planning. Here, timely approximate solutions are better than an ultimate solution, which may never come. The forte of the Genetic Algorithm is in finding these approximate solutions quickly.

2.1.4 DARWINIAN EVOLUTION

The Genetic Algorithm is inspired by the concepts used in neo-Darwinian evolution, and molecular genetics. Evolution, on the Earth, is a powerful force, shaping

using such methods, because, although the data in the search space is hierarchical, the search space itself has no external order placed on it.

✓

life for billions of years. During this time Evolution has 'solved' many problems that are of interest in Artificial Intelligence, and Engineering. Vision, sonar, motion planning, flight, structural designs that balance the requirements of light weight and strength (snail shells, egg shells, and nut shells), and of course intelligence itself, are just a few of the problems for which the solutions are, literally, embodied in animals, and other organisms, seen in nature.

The Genetic Algorithm is not the first time humanity has utilized Evolution to solve problems. Man, through agricultural breeding programs, has been using artificial selection to change many organisms into forms which more suit his needs. Dogs, cattle, wheat, and corn have all gone through substantial change over the millennia.

Evolution is obviously a powerful paradigm. It can also be seen as a search heuristic for optimization problems. Each organism is a potential solution to a 'problem' (flight, vision, etc.). How well the organism meets the problem is the organism's (the solution's) fitness. A better organism (solution) has a better chance to survive and reproduce, thus a better chance at being around during later generations. Less fit organisms tend not to survive long enough to reproduce, and so, through attrition, are not found in the population in the long run. Breeding and mutation cause variation in the population, which can be viewed as searching new points in the solution space. It is only a heuristic, since no mutation or breeding is guaranteed to produce an organism that is more fit than the others in the population for any given problem. Nor are the solutions in any given population maximally fit, and may never become so. However, the power of the evolutionary technique to find exceptional solutions, is without doubt. Thus, emulating Evolution should provide a powerful tool for searching large spaces, which is why it is the basis upon which GA is built.

2.2 THE GENETIC ALGORITHM

2.2.1 OVERVIEW

The Genetic Algorithm, expressed in its most basic form, simulates the evolutionary process of natural selection. A brief sketch of the algorithm is presented below.

First, a random set of potential solutions is generated. This is only done during the first iteration of the program. Each solution is, then, evaluated by the function which we want to maximize. The value, returned by the function, is called the '**fitness**' of the proposed solution, and the function itself is called the '**fitness function**'. A potential solution is called a '**chromosome**' or '**genotype**'. The set of chromosomes is called '**the population**'.

The GA, then, proceeds by repeatedly applying the processes of '**reproduction**'. Reproduction is the formation of a new population by a selected segment of the old population; either through direct copying, or by combining solution fragments, taken from the selected members, to form new solutions. Chromosomes are selected for reproduction in a probabilistic manner. The probability of selection must correspond to some function of the fitness of the chromosome. The various techniques for selection are called '**selection methods**'. This simulates the process of natural selection in nature; survival (or, more accurately, reproduction) of the fittest.

During reproduction, variation can be introduced into the population by changing a part of the chromosome to a random value, or by mixing two chromosomes together (combining the solution fragments), creating a new chromosome. These are called '**mutation**' and '**sexual crossover**' respectively (sexual crossover is also called

'mating' or just 'crossover'). These genetic operators are the main techniques used to produce new solutions in the population.

Finally, the new population replaces the old, and is evaluated using the fitness function. The algorithm now repeats itself, until a maximal solution is found, or an arbitrary (preset) number of generations have passed; at which point the best solution found during the successive generations is the answer given.

2.2.2 BIOLOGICAL TERMINOLOGY

The biological terminology used in GA is just there as an analogy and inspiration to the working of the algorithm. It does not necessarily duplicate, in function or form, the exact biological processes from whose theoretical description the terms are taken. Similarly, the results of the biological process may not be transferable to the GA equivalent. However, since there is a close correlation between biological evolution and Genetic Algorithms, and since enough of the analogies have proved to behave similarly, using biological terms is more helpful than misleading.

There are other biological terms commonly used in GA. They are used to describe features of the chromosome (see figure 2.1 for a diagram), and various analogous methods that are used (such as crossover and mutation), and similar properties observed in both systems (e.g. epistasis). The following terms will be used frequently throughout the rest of the thesis. A 'Gene' refers to the smallest part of the chromosome; usually the chromosome is represented as a bit string, so a gene would be a bit. In some cases a symbolic encoding is used, so the gene refers to a single symbol. The position of a gene in the chromosome is called the gene's 'locus' (in this paper, loci shall be enumerated starting from position 0). An alternative gene, to the one at a given locus, is called an

Figure 2.1 Chromosome Features

'allele'. One cycle of reproduction, and evaluation of the population is called a 'generation'. Finally, 'epistasis' is the dependence of the fitness contribution of one gene, on the value of another gene at a different locus. A high degree of epistasis can prevent the GA system from converging. The algorithm, as will be shown in section 2.4, gains its search speed through the independent, and concurrent, sifting of genes. Since epistasis prevents the genes from being independent of each other, no information can be gained by the system through the discovery of a good setting for any particular locus; after all, it may be a poor setting when the value of another locus is changed. Other biological terms in the paper will be explained when used for the first time.

2.2.3 ENCODING THE CHROMOSOME

Before starting the GA, the individual solutions, from the solution space, must be encoded in a form that the GA can process. The GA manipulation of its chromosomes is analogous to a biological system's manipulation of DNA strands. The DNA strands hold an encoding of the organism's body. The information is eventually expressed as proteins, by a decoding process at the ribosome (an organelle in the cell that converts genetic codes to proteins); thus producing the body. The encoded genetic material is called the organism's **genotype**, while the expression of the genotype as the body of the organism is called the organism's **phenotype**. The genotype of an organism is stored as a string of bases called **nucleotides**. There are four such bases: adenine, thymine, cytosine, and guanine². The DNA strand can be thought of as a machine language program, designed for the building of proteins (and ultimately the organisms that the proteins create), and written with a quaternary alphabet instead of the binary alphabet used for computers. Evolution, then, can be seen as the reproduction and manipulation of quaternary strings, moderated by the use of natural selection, which gleans information from the fitness of the decoded genotypes.

The Genetic Algorithm, just like Genetic Evolution, also encodes a phenotype, which is a potential solution, into a genotype, which is a string over a fixed alphabet. Also, like Evolution, strings can be reproduced and manipulated by the use of 'genetic' operators, analogous to those found in nature, such as sexual crossover, and mutation. As noted in section 2.1.1, a potential solution to an optimization problem corresponds to the appropriate input needed to maximize a function. A function takes a set of inputs, called the

² Nucleotides are often referred to as **base pairs**, since, in the DNA, they are always paired, with their 'opposite' base, in the second DNA strand: adenine bonds to thymine, and cytosine bonds to guanine.

domain, and maps it into a set of outputs, called the range. Since the input is a set of elements, they can always be encoded, sequentially, in a string (through simple enumeration, for example). Thus any potential solution, in the solution space, can be encoded into a genotype. Because of this encoding, a decoder has to be used on the genotype, to transform it back into a potential solution that can be evaluated by the fitness function.

Most Genetic Algorithms use a **binary alphabet** to encode the chromosomes. Larger alphabets are usually called '**symbolic alphabets**' since the large alphabet can be used to represent meaningful symbols, instead of the more difficult to interpret '0' and '1', of the binary alphabet. Binary alphabets, despite their cryptic form, were thought to have performance advantages over the symbolic alphabets when used in a GA. This has since been questioned (see section 2.4).

2.2.4 SELECTION METHODS FOR REPRODUCTION

Each new generation of the population is produced by the reproduction of the previous population. Reproduction involves mating, and straight forward copying, of the chromosomes from the previous generation. However, the mechanism that ensures improvement of the population over time, is the selection method. Selection, in GA, is the analog of Natural Selection in Darwinian Evolution. It is a probabilistic way of ensuring that the next generation is more fit than the current. Selection takes place during any type of reproduction. It selects the members to copy, and, selects the parents, if mating occurs. Selected members are returned to the population (sampling with replacement); so any member can be picked (and hopefully, if very fit, will be picked) more than once.

There are many different types of selection. The three most common are: Fitness-Proportional Selection, Rank Selection, and Tournament Selection.

Fitness-Proportional Selection

Fitness-Proportional Selection, also called **Roulette Wheel Selection**, is the earliest selection algorithm used. For Fitness Proportional Selection, an element is selected stochastically, with the probability of selection being proportional to the element's fitness, in comparison with the total fitness of all members of the population. In other words, all fitness values in the population are totaled, and the probability of selection for any element is equal to its fitness over the total fitness. If an element has a high fitness value, then, it has a greater chance of getting picked, rather than an element with a low fitness value. The expected ratio of the number of the element's offspring to the population size, is equal to the ratio of its fitness to the total fitness of the old population. An example of fitness proportional selection working on a population is given in figure 2.2. In the figure, the first pie chart shows the ratios of the fitness of each organism, to the total fitness of the population. The second pie chart shows the ratio of the expected number of offspring to be produced by each organism when Fitness Proportional Selection is used³. Notice the two charts are identical, since Fitness Proportional Selection produces an expected number of offspring in proportion to the fitness of the organism.

The reason that poor answer are sometimes selected is to keep the population diverse. While an answer may be bad in itself, a segment of it may be useful ,or even

³ The expected number of offspring, as shown, is really only expected if the new population formed is very large, with the old population selected from in numbers far greater than the size of the original population. The chart is also correct, if each pie segment is viewed as the expected number of offspring, of the member, over a *large number of trial experiments*.

Figure 2.2 The Effects of Fitness Proportional Selection

indispensable, to finding the correct solution, and all of the other better, although not perfect, answers might not contain it.

Fitness-Proportional Selection is an $O(n \log n)$ algorithm, where n is the population size. It is implemented by:

1. summing the population's fitness ($O(n)$);
2. finding (in a linear array) the total fitness to the left of each element, including the fitness of the element itself; thus creating a 'number line' whose divisions are proportional to each element's fitness ($O(n)$);
3. choosing a random number between 0 and the total fitness ($O(1)$);
4. finding which interval on the 'number line' the random number falls into ($O(\log n)$ if binary search is used).

The element corresponding to the interval chosen is the one selected. Steps 1 and 2 are done once for the population per generation, steps 3 and 4 are repeated until the new population is filled (the algorithm is $O(n \log n)$ since step 4 takes $O(\log n)$ and is performed n times).

Rank Selection

Rank Selection is similar to Fitness-Proportional Selection; the difference being that Rank Selection selects members of the population in proportion to their rank, when sorted by fitness, and not the value of their fitness itself (see figure 2.3 for examples). Rank Selection was created to remove a flaw implicit in the Fitness-Proportional Selection. Fitness-Proportional Selection runs into trouble when the members in the population are too similar in their fitness values, or too different. If one of the members in the population is only slightly better than the rest, the chances of it being picked are really no better than the others and so may not be selected, and thus can easily disappear from the population. On the other hand, if one member is far superior than all of the others, it will be overselected, and drive the system to a premature convergence. Rank Selection makes sure that slight advantages are seen and propagated, yet abnormally fit individuals are not overselected, because once a solution is ranked, the expected number of offspring in the next generation is fixed, irrespective of the size of the solution's fitness value.

The advantages of Rank Selection are shown, through examples, in figure 2.3. First notice that the various proportions of the expected number of offspring, after rank selection, are the same for the two populations (the pie size given to organisms of the same rank, are the same; they are just in different positions in the pie graph, based on the physical position of the organism in the population table).

In the first population all of the organisms have roughly the same fitness values. This is a common occurrence during the end of a run, when selection pressures have driven

Figure 2.3 Two Examples of the Effects of Rank Selection

the population to converge to good solutions, with only small performance differences between them. Notice that Rank Selection discriminates between the small differences, awarding more offspring to the higher ranking members. This allows the GA to continue to fine tune the solutions.

The second population is dominated by one abnormally fit individual. This can occur during the initialization process, where a very fit individual accidentally gets created. However, the number of offspring produced by this individual, seen in the bottom right chart, is far more restrained than would have occurred under Fitness Proportional Selection.

To find the rank, the population is sorted by fitness. Once sorted, the algorithm becomes similar to the Roulette Wheel Selection, with a function of the rank used as the probability mass, instead of the fitness. Since the rank is always the same, a probabilistic selection of an element, based on rank, can be derived in constant time, instead of using the $O(\log n)$ binary search, as is done for Fitness-Proportional Selection. The algorithm is implemented as follows:

1. Sort population according to fitness ($O(n \log n)$);
2. Choose a random number between 1 and the population size, ends included ($O(1)$);
3. Determine which element corresponds to the random number based on a function of the element's rank ($O(1)$).

Step 1 is done once for the population per generation, steps 2 and 3 are repeated until the new population is filled ($O(n)$). Like Fitness-Proportional Selection, Rank Selection is an $O(n \log n)$ algorithm (the time taken is dominated by the 'sorting' step).

Tournament Selection

The final selection method is the Tournament Selection. It is the least costly of the selection methods taking only $O(n)$ time. It is based on the tournament selection method found in nature, and sports leagues. For example, the selection of an alpha-male, in a group of chimpanzees, is done through individual challenges, producing a tournament pyramid; the final winner of the tournament becomes the alpha-male.

To select a member of the population using Tournament Selection, a subset of the population is chosen at random. The member of this sub-population that has the best fitness wins. The size of the sub-population is chosen by the user, but is frequently set to 2. The process, then, repeats, until the new population is complete. Tournament Selection has the advantage of being the least costly, and it also shares with Rank Selection the discrimination of closely distributed fitness values, and the prevention of the domination of an abnormally fit individual. It is a newer method, however, and its effects on the GA is not as well studied as the effects of the other two methods.

In all of the above methods selected members can be paired for sexual crossover. Also selected members are returned to the population (sampling with replacement). So any member can be picked (and hopefully, if very fit, will be picked) more than once.

Elitism

Because the selection routine is stochastic, the best solution in the population, for a given generation, may not be selected, and so may not be passed on to the next generation. This is to be expected, and, to some extent, is not a problem, since the population base, that formed the solution originally, is still there, and can form another solution as good, or better, than the one that was lost. However, the good solution that was lost, could have been a good stepping stone for the formation of even better solutions; so a selection method called 'elitism' is sometimes implemented.

Elitism selects a sub-population of the most fit members of the current population, and passes them on, untouched, into the next generation. The number of members that are selected (called the elite) is determined by a user defined parameter. Elitism, in practice speeds up the rate of convergence for many GA applications, and is frequently used in many GA implementations ([Davis 91] pg. 34).

Elitism has detrimental effects as well. While it speeds convergence, it can also lead to premature converge, by fostering the dominance of an abnormally fit individual: that individual soon becomes the only genotype in the elite group, giving it an unbeatable advantage, and out breeds all competitors. A modification of elitism, called unique elitism solves this problem (in Davis it is called "steady state without duplicate"⁴). Under unique elitism, only one individual with the same genotype is allowed to be passed to the next generation as an elite. This does not necessarily prevent premature convergence, after all premature convergence can occur in a system with no elitism at all, but it minimizes the advantage that a 'super' individual can attain through elitism.

2.2.5 THE FITNESS FUNCTION

The term 'fitness function' is not usually used in reference to the optimization function being maximized by the GA. There are two reasons for this. The first reason is caused by restrictions placed on the fitness function to facilitate the use of a selection method; the optimization function, usually called '**the objective function**', may not

⁴ Davis uses the term steady state for a system that does not form a new population, but replaces the most unfit individuals in the population with the reproduced individuals selected. However this really is the same as elitism, since, when using elitism, the new population has the all the elite members from the old population. Therefore the new population after reproduction with elitism, contains the same individuals as the old population would after reproduction in a steady state system.

follow those restrictions, and may need a 'wrapper' to convert it to a proper form. This is especially true with the use of Fitness-Proportional Selection, which needs monotonically increasing fitness, and non-negative values. The second reason to modify the fitness function, is to enhance the dynamics of the GA, helping the algorithm converge to a better solution (or the correct solution) faster. When the fitness function used is a modification of the objective function, if the objective function has a known maximum, the fitness function is often designed to return a number between 0 and 1.

Most of the restrictions on the fitness function are only required by the Fitness-Proportional Selection method. The Fitness-Proportional Selection method only works with the maximization of the fitness function. Furthermore, the fitness function must be non-negative. (Both the Rank, and Tournament, selection methods can be easily modified to handle minimization objective functions, and negative numbers). If the objective function is naturally a minimization problem, negate the function, and shift it above zero by adding the absolute value of a known, global minimum. If the global minimum is not known, shift the negated function by adding the current minimum, or the best minimum encountered so far. If the objective function measures a minimization problem that has a range of 0 to infinity, then, it can be transformed to a maximization problem with a range of 0 to 1 by the following transformation:

$$f = 1 / (1 + u)$$

where u is the evaluation of the objective function, and f is the converted fitness function value. This fitness is sometimes called 'the adjusted fitness'.

2.2.6 CROSSOVER AND REPRODUCTION

At first glance, it might seem strange that poor members are kept, during reproduction, even if it is with low probability. They are kept, because, even though the entire answer may be unfit, parts of the answer might be useful, or even necessary to the eventual solution. The function that mixes the different parts of the solution is called crossover.

Crossover is based on the sexual crossover as observed in nature. Two organisms of the same species, called parents, splice segments of each other's chromosome together, during reproduction, to form a 'new' chromosome. The new chromosome is used as the genome of an offspring. The two chromosomes (one from each parent), during crossover, are matched, so that each locus is aligned. This makes sure that the swapped genetic sequences have the same meaning, or, in other words, have an effect on the same phenotypic feature when decoded.

The crossover, in GA, works analogously. Two parents are selected, for mating, by a selection method. In some implementations, both parents are selected using the same method; in others, only the first parent is so chosen, the second parent is selected randomly from the population. An offspring is then produced, by taking chromosome segments from each of the parents, making sure that each locus is represented once and only once in the new chromosome. Usually long segments from a single parent are kept together. Two children are produced from the crossover operation, with each locus being from the opposite parent (for examples of various crossover operators, see figures 2.4 & 2.5).

Figure 2.4 1-Point Crossover

Since selection occurs with replacement, the same gene can be used for both parents. When this happens crossover has no effect, and the result is as if simple reproduction were used.

The locus, in an offspring, where the genetic sequence from one parent ends, and the other parent begins, is called a 'crossover point'. Many GA implementations use one or two randomly selected crossover points for the crossover operation. This is called '1-point crossover', and '2-point crossover', respectively (for an example of 1-point crossover see figure 2.4). When 1-point crossover is used, if the crossover point is chosen at either end, crossover has no extra effect aside from reproduction.

Another popular crossover technique is the **uniform crossover** [Syswerda 89], [Spears & De Jong 91]. The uniform crossover copies the two parent chromosomes into their respective children, but decides at each locus whether to swap the genes. The probability that swapping will occur at a given locus has a uniform distribution across the loci. In other words the probability that swapping will occur at a given locus is the same for all loci. This probability is an external parameter set by the user.

Crossover is not applied for all chromosomes selected during reproduction. The chromosome may be copied unchanged, or have some other genetic operator applied to it.

A Gene from Parent 1 / 2, based on the Crossover Template, gets copied into Child 1 / 2 or crossed into Child 2 / 1

Legend: C Copy
X Cross

Figure 2.5 Uniform Crossover With a 0.5 Probability for Gene Exchange

The frequency of crossover occurring during reproduction is an external parameter of the program.

Crossover is a vital part of the Genetic Algorithm. Early attempts at GA ignored this aspect of biological evolution, relying on mutation and selection alone, and failed. The reason that crossover is so important is not fully understood yet, but some insights have been developed, and will be examined in section 2.3.

2.2.7 MUTATION

Another major genetic operation, used during reproduction, is mutation. Like crossover, it is inspired by the biological analogy of genetic mutation. Here, during reproduction, a gene is accidentally replaced by an allele. For example, at a locus that is

occupied by the base guanine in the parent, the base thymine is found in the child. This (along with crossover) allows for variation in the offspring, which natural selection can act upon. Crossover explores different combinations of existing alleles in the gene pool, while mutation introduces new alleles, at a locus, to the gene pool.

Mutation, in GA, is completely analogous to genetic mutation. During mutation, a gene, at a random locus, is changed to an allele. When each locus is copied, during reproduction, there is a chance that it may mutate. The probability that a given gene will mutate is called the '**probability of mutation**', or the '**mutation rate**'. Since only one locus is affected by a single mutation, the operator is often referred to as '**point mutation**'. The mutation rate is the second external parameter of the genetic algorithm.

Mutation is not as important as crossover in the genetic algorithm. Consequently the mutation rate is set very low (as it is in nature). A common setting for the mutation rate is 0.001, which means one gene, out of every thousand loci copied, is expected to mutate.

Infrequently as it occurs, mutation is still vital to the convergence of a GA. This is true, because an allele that is necessary to the correct solution may be bred out of the population prematurely. Mutation allows alleles, that have been eliminated, to creep back into the population.

2.2.8 CONVERGENCE, AND A POPULATION OF SOLUTIONS

Observing a set of possible solutions (the population), instead of one solution at a time, as used in most search techniques, is patterned after the concurrent 'evaluation' of biological organisms in the environment. The GA is, therefore, inherently, a parallel algorithm, implemented sequentially. The use of a population of solutions is the mechanism which prevents the GA from returning a local maximum. Since the population

is spread out over the solution space, the chance that all solutions will be attracted to the same local maximum is smaller than in normal hill climbing (mutation + natural selection can be thought of as a hill climbing technique). Also, the result of a crossover between two local maxima, may try areas that are outside the convergence zone of either local maxima. This is not to say that Genetic Algorithms will always give the global maximum. Being a probabilistic algorithm, a GA may not converge within a reasonable time, or at all. A solution that is significantly better than the others in the population can, over time, become too prevalent in the population, stifling diversity, and forcing a premature convergence. Instead of having the advantage obtained by the use of a large population, the abnormally good (but not optimal) solution becomes, in effect, the only solution, reducing the population to one individual. There are techniques to prevent this. Rank Selection is one such technique, which works well. So, although any specific run may not converge, the GA is designed to produce answers that approach the global maximum over time.

The size of the population is very important for the convergence of the GA. If the population is too small, then not enough alleles get produced for the crossover operator to sift through. Mutation will, then, have to take over, which means that the GA is reduced to blind search. This slows down the convergence to a point where the GA might not find the answer in any realistic time. Too large a population wastes resources, and may not produce the answer any sooner than a smaller population would, and so just slows down the algorithm. Population size is the third parameter set by the user, and is usually problem dependent. Many GA systems use a population size of 40 to 100, however some applications, such as Genetic Programming, use population sizes of up to 1000, or more.

2.2.9 TERMINATION CONDITIONS

If the maximum fitness is known then the algorithm stops when the maximum fitness is reached. However if the maximum fitness is unknown, or is approached but not reached, the iteration can continue forever. In biology the iteration never does complete, since life is a never ending process, and does not have a goal in mind. However, a non-terminating algorithm is not desirable in a computer program. In practice, the GA is only run until a given generation. At that time, the best solution found, so far, is returned as the result of the search. The GA can stop earlier, if the maximum fitness is known, and is encountered before the final generation is reached. The number of generations to be run before termination, is the fourth parameter in the GA, and is also externally set by the user. It is the least important of the parameters, in terms of the workings of the GA

2.2.10 PUTTING IT TOGETHER: THE GENETIC ALGORITHM

The following is a top level overview of the genetic algorithm:

- 1) Randomly Generate the Population of Chromosomes (called Population)
- 2) Decode the Chromosomes of Population into their phenotypes
- 3) Evaluate the Fitness of the Population
- 4) Loop for gen generations or until Max fitness found
(if there is a known Max fitness)
 - 4a) Create a new generation from the old population using a selection method to select the members, and reproduce them using the genetic operators
 - 4b) assign to Population, the new population
 - 4c) Decode the Chromosomes into their phenotypes
 - 4d) Evaluate the fitness of Population

Most of the work in the algorithm is performed in step 4a. This step can be very different from one implementation to another, depending on which genetic operators are chosen. An expansion of step 4a using 1 point crossover is shown in figure 2.6.

```

Loop for size of population, pop-size, by 2
  parent1 <- select (population)
  parent2 <- random (population) or select (population)
  If random number < x-prob, (the crossover probability)
    Randomly select a crossover point (xover-pt)
    For child 1:
      Copy loci from parent1 beginning at locus 0 below xover-pt
      Copy loci from parent2 beginning at locus xover-pt below chr-length
    For child 2:
      Copy loci from parent2 beginning at locus 0 below xover-pt
      Copy loci from parent1 beginning at locus xover-pt below chr-length
    else
      copy parents
  end if
  insert the children into the new population
end loop

```

Note: During the creation of any chromosome copy, either through crossover, or direct reproduction of a chromosome, each locus has a **m-prob** chance of mutation

Legend:	x-prob The Probability of Crossover m-prob The Probability of Mutation xover-pt The Crossover Point gen The Maximum number of Generations chr-length The Chromosome length pop-size The size of the Population random (population) Select a random member from the population select (population) Select a member from the population, using a selection method
---------	---

Figure 2.6 An Expansion of Step 4a Using 1-Point Crossover

The user of a GA has control of various parameters and design features, which allows the GA to be tailored to an individual problem. There are three levels of control: parameter settings, system design choices, and application routines.

The simplest of the control structures, in a GA, are the system parameters. The system parameters control the various mechanisms in the GA, such as the population size, and mutation rate. They have been presented throughout the chapter, when they were applicable. A summary of these parameters is in table 2.2.

The second level of control is the system design choices. Most GA systems come with a variety of different design features, which can be selected from. These include the various crossover techniques, and selection methods, which have been introduced throughout this chapter. A summary of these design options can be found in table 2.3.

The third level of control, in a GA, is the set of procedures that tailor a GA to solve a particular application. Before the actual procedures are written, they should be carefully designed. Design considerations includes: choosing a chromosome encoding of the problem to be solved; choosing an alphabet for the encoding⁵; and designing a fitness

Table 2.2 User Defined Parameters

Parameter	Description
x-prob	The Crossover Probability
m-prob	The Mutation Probability
gen	The Maximum Number of Generations
chr-length	The Chromosome Length
pop-size	The Size of the Population

⁵ A binary alphabet is used in most GA systems, and are applicable to most problems, but sometimes symbolic alphabets are more appropriate, such as for a program induction application.

Table 2.3 Design Options

Procedure	Procedure Option	New User Defined Parameter
Crossover Operators	1-Point Crossover	None
	2-Point Crossover	None
	Uniform Crossover	Probability of Gene Exchange
Selection Methods	Fitness-Proportional Selection	None
	Rank Selection	Expected Number of Offspring for Top Ranking Member
	Tournament Selection	Number of Elements in the Tournament Set
	Elitism	Percentage of Elite in Population

function which both produces a good evaluation function, isolating the proper feature to be maximized, and is compatible with the selection method used. The actual information, and procedures passed to the GA are: the alphabet used in the chromosome, the fitness function, and a decoding function to transform a chromosome into a potential solution that can be evaluated by the fitness function. With these elements in place, the GA can solve a wide range of problems, with greater speed and flexibility than most other comparable systems.

2.3 GENETIC ALGORITHM THEORY

The basic theory, explaining how GA can be so successful in finding solutions in an exponentially large search space, was developed in the early seventies, by John H. Holland [Holland 75]. Holland's terminology, and methods, are a little outdated now, and

so can be hard to read. A concise, and clearly written, summary of GA theory can be found in [Goldberg 89] (all definitions and equations presented in this section come from Goldberg).

The analysis is based on the effects of a GA which uses fitness proportional selection and 1-point crossover (both of which were developed by Holland and presented in the same book) on the chromosomes in a population, and, in particular, on the schemata in the chromosomes. A **schema** is "a similarity template describing a subset of strings with similarities at certain string positions" ([Goldberg 89] pg. 19). For example, using a binary alphabet, with an added symbol '*', which means "don't care" (matches any symbol from the alphabet), then, '11*0**1' is a schema that matches the chromosomes '1100101', '1100001', '1110011', among others. In fact there are $2^3 = 8$ chromosomes that match the above schema. In general for a schema H , there are $k^{\vartheta(H)}$ chromosome which match it; where k is the number of symbols in the alphabet, and $\vartheta(H)$ is the number of fixed positions (i.e., the number of loci in the chromosome, minus the "don't care" positions). Since each locus can either be a symbol, or a "don't care", there are $(k + 1)^l$ schemata that can be expressed in a chromosome of size l .

The governing equation for the behavior of the GA system is:

$$m(H, t+1) \geq m(H, t) \frac{f(H)}{\bar{f}} \left[1 - p_c \frac{\delta(H)}{l-1} - \vartheta(H) p_m \right];$$

where $\vartheta(H)$ is the number of fixed positions; $\delta(H)$ is the number of loci between the first fixed position and the last fixed position, including end points (e.g., $\delta(11*0**1) = 7$ and $\delta(**1*0**) = 3$); $f(H)$ is the average fitness of all chromosomes that match schema H ; $\bar{f}$ is the average fitness of the entire population; p_c is the probability of crossover; p_m is the probability of mutation; l is the length of the chromosome; and $m(H, t)$ is the number of chromosomes that match schema H in the population, at time t . The equation models the number of chromosomes, which match schema H , in successive generations. The core of

the equation is the effect of fitness proportional selection on the number of chromosomes which have schema H embedded in them. This is modeled by the $\frac{f(H)}{\bar{f}}$ term. The effects of the selection method is modified by disruptions to the schema from the genetic operators: $p_c \frac{\delta(H)}{l-1}$ is the adjustment term, in the equation, for the 1-point crossover, and $\mu(H)p_m$ is the adjustment term for mutation.

The effect of fitness proportional selection guarantees that "reproduction allocates exponentially increasing (decreasing) numbers of trials to above- (below-) average schemata" ([Goldberg 89], p. 30). The disruption caused by the crossover and mutation, restrict the types of schemata that can have their numbers enhanced in the population; favoring the survival of short, low-order, schemata. The effects, together, lead to the '**Schema Theorem**': "Short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations" ([Goldberg 89], p. 33). In other words, it is the "highly fit, short, low-ordered schemata" that are favored by GA. These types of schemata are called "building blocks". If this analysis is correct, then, when encoding the potential solutions to a problem into a chromosome, they should be designed so that the building blocks are "relevant to the underlying problem and relatively unrelated to schemata over other fixed position" ([Goldberg 89], p. 80). This is called '**the principle of meaningful building blocks**'.

Using the Schema Theorem, the number of *useful schemata* that are processed in a population of n chromosomes has been calculated as being $O(n^3)$ (for a binary alphabet, with n chosen to be equal to $2^{l_s/2}$, where l_s is the length of the largest schema that can statistically survive during reproduction). In other words, there are roughly n^3 useful schemata processed, when n chromosomes are processed. This notion, that many schemata are being sifted through, and evaluated, at one time, is called '**implicit parallelism**'.

The principle of meaningful building blocks is a guideline, based on the Schema Theorem, to be used when encoding, as a chromosome, potential solutions to the problem being solved. Another encoding principle, 'The principle of minimal alphabets', has been derived from the Schema Theorem, and is used by many GA applications. The principle states: the smaller the alphabet used, the better the GA will work. This conclusion is based on the number of schemata presented to the system; the more schemata present, the more information the GA can use to find the correct answer (remember, a GA works by processing schemata in parallel; the more schemata present, the more work can be done in parallel). The number of schemata is $(k + 1)^{l/\lg(k)}$, where k is the number of values in the alphabet, l is the chromosome length when encoded in a binary alphabet, and $\lg$ is the logarithmic function, base 2. This number is maximized when $k = 2$, therefore, when binary alphabets are used, the greatest number of schemata are present in the system.

There are difficulties with the basic GA theory, presented above. While it presents good insights into the workings of the GA system, it does not provide a complete framework, and so has had to be extended. It is based entirely on fitness proportional selection, and the one point crossover. However, there are many other successful selection methods, such as Rank Selection, that have different properties than fitness proportional selection, and so are not covered by the theory as it stands. This was rectified by Grefenstette and Baker, who modified the analysis to reflect the effects of the broadest possible class of genetic algorithms [Grefenstette & Baker 89]. However the description thus produced, rather than be too exclusive, is too inclusive. Work on this problem is an active area of research in the field.

The principle of minimal alphabets has also come under attack [Antonisse 89]. The schema counting used by the argument, ignores some cases. The "don't care" symbol '*', was used to mean 'any symbol from the alphabet'. However, it might also be interpreted

as a place holder for any set of symbols (with a cardinality greater than 1), from the power set⁶ of the alphabet used. For example (from Antonisse's paper) a schema '000*', encoded with the alphabet {0 1 2}, would, in the earlier interpretation, represent the chromosome set {0000 0001 0002}. However, with the new interpretation, it corresponds to the sets {0000 0001}, {0000 0002}, {0001 0002}, and {0000 0001 0002}⁷. In other words, the schema 000* could mean "'don't care" if it is a 0 or a 1, but it is definitely not a 2', or equally, "'don't care" if it is a 1 or a 2, but it is definitely not a 0', as well as the original meaning "'don't care" if it is a 1, 2, or 3'. These cases are not considered in the original accounting. Therefore, the recalculated number of schemata, in a chromosome of length l , is $(2^k - 1)^l$ for an alphabet of size k . This is derived from the number of possible values at each locus, which is the size of the power set without the Null set, multiplied together, once for each locus in the chromosome. When recalculated, an n -ary alphabet can actually represent more schemata than the binary alphabet. For an example, compare a binary alphabet, on a chromosome with 8 loci, with a ternary alphabet, on a chromosome with 5 loci (this gives, approximately, the same amount of information encoded). The number of schemata for the binary alphabet is $2^8 - 1 = 6561$. However, the number of schemata for the ternary alphabet is $(2^3 - 1)^5 = 16897$, not $2^5 - 1$, as would have been computed previously. This is larger than the number of schemata computed for the binary alphabet, above. For further analysis on the advantages of an n -ary alphabet, see section 5.3.1.

For all the faults of the Schema Theorem, it does present a useful first attempt at understanding how a GA works. As time progresses, perhaps a more complete theory will be developed. Until then, the Schema Theorem at least provides a theoretical framework to

⁶ A power set of set 'A' is the set of all subsets that can be formed from 'A'.

⁷ The power set of the alphabet {0, 1, 2} is the set:
 $\{ \emptyset, \{0\}, \{1\}, \{2\}, \{0, 1\}, \{0, 2\}, \{1, 2\}, \{0, 1, 2\} \}$

support the experimental evidence that the GA system is a good search heuristic for solving optimization problems.

CHAPTER III: A BRIEF REVIEW OF PROGRAM EVOLUTION

3.1 INTRODUCTION

Program evolution is a natural extension of GA. Instead of evolving a single solution to a task, why not evolve a general solution to the task itself. However, there were problems, that were evident early on, that prevented the development of a program induction application on a GA. Programming languages, as traditionally used by humans, are very brittle in their structures. Small changes in the code, can change a correct program, into a program that is wildly wrong, or even syntactically incorrect.

3.2 CONTEXT SENSITIVE INTERPRETATION AND ORDER DEPENDENCY

The problems, inherent in procedural languages (which is the style of programming language most familiar to programmers), have been summarized by De Jong [De Jong 87], [De Jong 89]; they are: order dependencies, and context sensitive interpretations. **Order dependency** occurs when the position of program structures are important. When, for example, statements in a program must be written in a specific order, or the parameters for a function can not be interchanged, without the program becoming meaningless, the program is said to have order dependencies. **Context sensitive interpretation** occurs when the meaning of a value is affected by changes in another part of the program (this is analogous to epistasis).

It was thought that these problems would arise often in GA, for a program induction application, because of the use of blind genetic operators, such as mutation and crossover, which would not be careful of the order dependencies, and context sensitive interrelations, of the programs encoded in the chromosome. For this reason production

systems were chosen for **Classifier Systems (CS)**, the first machine learning algorithm developed for GA [De Jong 89]. A **production systems** is a language, composed of a series of if-then clauses that produce a plan to solve a task. Production systems are 'Turing complete', that is, on a machine with unlimited memory, they can solve any problem that the 'Turing Machine' can, and therefore can be considered a programming language. However, production systems, like most programming languages, can more readily solve certain types of problems than others; consequently CS would be more applicable to certain applications than to others. Therefore, another system, which may be adept at solving different types of problems, is enticing.

3.3 PROGRAM EVOLUTION SYSTEMS

There have been various attempts at using an explicit procedural programming language, as the basis of program induction through evolution. These began a decade after GA was introduced, and half a decade after the first CS was produced. This eventually lead to Koza's 'Genetic Programming' by the late eighties, and early nineties, which is the current, 'state of the art', system.

3.3.1 JB AND TB

In 1985 Michael Cramer proposed using a small, although 'Turing complete', language, called PL as the basis of a GA that evolves programs, called JB[Cramer 85]. PL has the following operators: **(:inc var)**, which increments the variable 'var'; **(:zero var)**, which sets 'var' to 0; **(:loop var stat)**, which evaluates a statement, labeled 'stat', 'var' times; and **(:goto lab)**, which jumps to the label 'lab'. All PL operators, except the

:goto operator, were used in the GA. Each PL statement was encoded as a numerical triple. The triples are placed sequentially to produce the chromosome. The JB system used these chromosomes to perform program induction using a normal GA, with the standard crossover, mutation, and selection methods.

However, JB had problems: infinite loops would easily form; and, as a result of PL's vulnerability to context sensitive interpretation, high epistasis was observed when the genetic operators were applied.

Cramer then tried to use a variation of the structure of the PL language, as the basis for the chromosomes in JB, which was renamed TB. TB produces a tree like structuring of the program, instead of proceeding linearly through the statements. This proved to solve most of the problems, although mutation had to only be applied near the leaves, since they still lead to "catastrophic minor changes" when used freely. In order to implement a form of crossover that could handle the tree structures of TB, crossover was only allowed to exchange subtrees between two parents¹.

Cramer's system is a very interesting first attempt to implement program induction as a GA. In her paper, she uses the system to solve the two input single output multiplication problem. However, because the language it uses is so low-level, it is difficult to see how it could solve very complex problem, without significant changes to increase the power of the algorithm.

¹ Cramer in her paper does not make it clear whether the crossover point must have the same locus in both parents, which would make it analogous to the two point coupled crossover introduced in section 5.5.3, or whether the exchanged subtrees are from different loci, in which case it would be more analogous to the crossover used in Genetic Programming.

3.3.2 A GA THAT MANIPULATES LISP CODE

In 1987, Fujiki and Dickinson used the LISP programming language, as the basis of a GA chromosome encoding, to create programs which solve the Prisoner's Dilemma problem [Fujiki & Dickinson 87]. However, the system created did not use the full nested list properties of LISP. The encoding of the chromosome was restricted to a subset of LISP, the 'COND' statement. 'COND' is a linear, not tree like, structure, that is a sequence of condition/action pairs. This makes Fujiki and Dickinson's system of encoding similar to the CS production system. A chromosome can be of variable length, since the 'COND' statement can consist of any number of condition/action pairs. The crossover used is similar to the 1-point crossover, except that the crossover points chosen may be different for the two parents. A type of inversion operator, which reverses the order of a sequence of condition/action pairs, was also used.

3.3.3 GENETIC PROGRAMMING

In 1989 John R. Koza introduced Genetic Programming (GP). This is the most successful attempt, at evolving programs, produced to date. It is a combination of the tree structure and subtree crossover from TB, and the chromosome encoding of LISP functions from the system by Fujiki and Dickinson. Unlike Fujiki and Dickinson, GP uses the full tree structure, and evaluation ability, of the LISP programming language.

At the heart of GP lie two basic insights: the LISP s-expression is equivalent to a parse tree; and different, problem specific, primitives, which satisfy the property of closure, should be used for each new task to be solved. These properties help alleviate the problems of both context sensitive interpretation, and order dependency. The use of

LISP s-expressions as parse trees, addresses the problem of context sensitive interpretation (at least syntactically). Since a parse tree is fully modular, any change to its structure, as long as the integrity of the tree is kept, will produce valid programs. Order dependencies are alleviated by the use of closure in the formation of the primitives used by the system. Closure ensures that the output of any operator, or any primitive operand, is valid as input for any of the operators. This allows changes to be made in the order of the program, and still the program remains syntactically correct. Although changes to the order of the subtrees will still affect the results of a program, the programs will still run. The radical changes to the program structure that result from crossover and mutation should not cause concern, since it is not very different from changing a gene to an allele. Changes to the order of the subroutines that are beneficial to the individual, will be favoured by the GA, and, those that are detrimental, will die out.

The use of specially designed primitive operators and operands, also allows GP to implement far more complex tasks than if more general operators are used, such as those in JB, and TB. This is because the primitives embody large amounts of information about the problem to be solved. This information would have to be developed, through evolution, if general purpose primitives are used. This would slow the system down tremendously. Since Genetic Evolution is formed from general primitives, and yet has 'solved' very large, and complex, tasks, it is not impossible that a computer system could do likewise. However, Evolution took Billions of years to develop the 'solutions', and it constantly builds on past results. Recent organisms evolve much more quickly (by orders of magnitude) than did the primitive life forms from which they emerged. By using complex primitives, GP 'jump starts' the system, and need not wait the billion years that may be necessary to evolve solutions to complex tasks.

The system developed in this paper builds on the design of Genetic Programming. Therefore, a full description of GP is given in the next chapter.

3.4 SUMMARY

There is one common feature in all three systems presented above. In order to have a GA that can solve program induction, non-traditional genetic operators and structures were introduced. It has been hypothesized that generalizing the GA to use formal languages will necessitate the use of new crossover operators [Antonisse 89]. The goal of this paper is to try and develop a system that leaves the traditional genetic operators intact, yet still succeeds in evolving programs.

CHAPTER IV: GENETIC PROGRAMMING

4.1 INTRODUCTION TO GENETIC PROGRAMMING

Genetic Programming is an extension of the Genetic Algorithm, introduced by John R. Koza in 1989. GP uses a modified GA to evolve a population of programs. Since it is the most successful program induction application of a GA-like system, and since many of the ideas used for the EP-I system are either derived from, or contrasted to, GP, it is useful to describe GP in more detail than was done in Chapter III. A brief overview of the ideas behind GP is first presented, followed by a complete description of the algorithm. This is followed by a section detailing the functional differences between GP and a more traditional GA.

The sections 4.5 and 4.6 are concerned with the experimental side of GP. The first of these sections presents the statistics used when designing experiments which determine the chances of an experiment converging to an answer. The second is a description of the test suite of applications that are used in the thesis. This includes: an overview of the problem to be solved; a summary of the initial settings, used in GP, for each application; and the probability of convergence, as reported by Koza, which rates how well GP solved each of the problems.

Finally, the last section has a brief discussion on a proposal, by Koza, for a new schema definition for the GP tree structures, which, Koza suggest, allows the Schema Theorem to be applied to GP, thus giving GP a mathematical foundation to support its success. Counter arguments are presented demonstrating that the analysis is incomplete, at best, and, possibly, of no value.

For more information on Genetic Programming, and for many (many) more examples of applications that can be solved by GP, read "Genetic Programming: On The Programming Of Computers By Means Of Natural Selection" by John R. Koza.

4.2 THE USE OF LISP S-EXPRESSIONS AND PARSE TREES

4.2.1 PROGRAMS AS CHROMOSOMES

In order to have a workable GA that evolves programs, one must be able to encode the programs as chromosomes, which, when manipulated by crossover, will allow good segments of the genetic code to be brought together. Using a naive encoding and crossover, however, there is no guarantee that the resulting children will be syntactically correct programs. Genetic Programming gets around this problem by encoding the program as a parse tree.

4.2.2 PARSE TREES AS CHROMOSOMES

A **parse tree** is a rooted, n-ary tree, which has operators as internal node values, and operands, called **terminals**, as leaf values. Parse trees can be thought of as nested functions explicitly represented in tree form. These functions can be evaluated to produce an answer. This is called '**collapsing the tree**'. The tree is collapsed by recursively collapsing subtrees that only have a single internal node. A subtree with only one internal

```

collapseTree (tree)
  if root (tree) = a terminal
    result := root (tree)
  else
    operator := root (tree)
    loop for i = 1 to childrenCount (tree)
      subtree := subtree (tree, i)
      add collapseTree (subtree) to arg-list
    end loop
    result := apply (operator, arg-list)
  end if
  return result
end collapse-tree

```

Legend:

<i>root (tree)</i>	returns the value at the root of the tree 'tree'
<i>childrenCount (tree)</i>	returns the number of children of the root
<i>subtree (tree, i)</i>	returns the i th subtree of 'tree'
<i>apply (operator, arg-list)</i>	applies the operator 'operator' to the arguments in 'arg-list', and returns the result

Figure 4.1 A Recursive Algorithm for Collapsing a Parse Tree

node is collapsed by applying the operator, taken from the internal node, to the terminals, and replacing the entire subtree by the result, which is now a terminal. The process continues, until the final answer replaces the root. The tree is now completely collapsed, and the final answer is returned. Collapsing the tree can be written as a simple recursive algorithm (see Figure 4.1).

All programs can be written as a parse tree. When a parse tree incorporates operators that have side-effects, the order that the tree is collapsed becomes important. Usually the *collapseTree* algorithm from figure 4.1 is used, which is a depth first tree traversal. However, the order of traversal can be changed, which, if it occurs, will affect the results of the side-effects. In some systems, the operators can determine the order of evaluation of their children, or, even, the suppression of a child's evaluation.

Figure 4.2 GP Crossover Example for Parse Trees

Since each subtree is a valid parse tree in its own rights, a simple crossover technique can now be defined. Two parse trees are selected to be the parents. Both parents are duplicated, producing the basic genetic code for the two offspring. In each child, a random node is chosen. These are the crossover points. Notice that they do not have to have the same position in the two trees. In fact, since parse trees can be of different shapes, the notion of nodes with comparable positions in the two trees, is ill-defined. The crossover points define the roots of two subtrees. The subtrees are exchanged, completing

the crossover operation. The two children now have different genotypes from their parents. A simple example is shown in Figure 4.2.

One obvious observation from the example above is the variation in the tree heights between the parents and the children¹: Parent 1 has a height of 3, and Parent 2 has a height of 4, while Child 1 has a height of 5 and Child 2 has a height of 2. This occurred, because, the two nodes, picked for crossover, are in different levels in their respective trees. This is a very different situation than when crossover is applied in the regular GA; the crossover point has the same location in both of the parents' chromosomes, thus producing children with chromosomes of the same length. As noted before, the selection of two different points for crossover is necessary in GP since the two parents may have vastly different structures, making the selection of a single crossover point meaningless.

4.2.3 LISP PROGRAMS

Programs, in any programming language, can be converted to, and from, parse trees. However, the programming language LISP is a natural choice to use for Genetic Programming. This is because LISP has a simple, one-to-one, mapping to parse trees.

LISP uses nested lists to describe a program. A list, that represents a program, is called a **symbolic expression**, or '**s-expression**', for short. The first element, in the s-expression, is called the **function**, the rest are called the **arguments**. This ordering, with the function presented first, then the arguments, is called **prefix notation**. LISP s-expression can be nested; each of the arguments of an s-expression can be s-expressions themselves. The evaluation of an s-expression proceeds as follows: the arguments are

¹ Level counting, in a tree, begins with 0 for the root level. Therefore a tree with only a single node in it, has a tree height of 0.

Figure 4.3 Components of LISP S-Expressions

evaluated first; then the function is applied to the returned values. The process is recursively applied when evaluating the arguments. This evaluation procedure is identical to the *collapseTree* procedure, used on parse trees in the previous section, with the list replacing the tree. In fact a tree can be represented by a nested list. So a LISP s-expression is equivalent to a parse tree.

LISP also facilitates symbolic manipulation. This makes it easy to have a LISP program produce, manipulate, and evaluate other LISP programs. Using LISP's list processing capabilities, a valid LISP program can be put together, and, then, evaluated as a regular LISP program. So instead of having to convert a program into a parse tree in order to manipulate it, the population of LISP programs can be directly manipulated and evaluated.

The arguments of a LISP s-expression do not have a pre-defined order of evaluation. Consequently the order of evaluation is dependent on the implementation.

```
(defmacro foo (arg1 arg2 arg3)
  `(let* ((arg-c (eval ,arg3))
         (arg-b (eval ,arg2))
         (arg-a (eval ,arg1)))
    (bar arg-a arg-b arg-c)))
```

The macro 'foo' takes three arguments, evaluates them in reverse order, then applies the function 'bar' to them.

Figure 4.4 A LISP Macro That Reverses the Evaluation Order of its Arguments

Many implementations evaluate the arguments from left to right, thus traversing the corresponding parse tree in a depth first order, just as the *collapseTree* procedure does. There are some functions that have an enforced depth first order such as 'and', 'or', and 'progn'. If a pre-defined order of evaluation is necessary because of side-effects, then the LISP macro facility can be used to create 'functions' that evaluate the arguments in the order the user requires. A macro does not evaluate its arguments, but passes them, by name, into the body of the macro. This allows the user to write code, for the body of the macro, that evaluates the arguments in any order they wish (see Figure 4.4). This also provides a mechanism to perform conditional evaluation of sub-expressions, if needed.

4.3 THE GENETIC PROGRAMMING ALGORITHM

4.3.1 OVERVIEW OF THE ALGORITHM

Genetic Programming begins with a set of pre-defined operators and terminals, and an initial population of randomly generated solutions. The creation of the initial population is an initialization step, and is only done at the start. Once the initial population has been created, each program is evaluated, to see how well it operates in comparison with known output. The creation of a new population now begins. Using the results of the fitness

function, programs are selected by a standard GA selection method. Reproduction of the selected programs is done by either copying a program, or mating two programs together using the crossover operator for parse trees. Reproduction continues until the new population is complete. After reproduction, the new population is evaluated, and the algorithm begins the production of the next generation. This cycle continues until an acceptable program is created, or until a maximum number of generations have passed.

4.3.2 THE OPERATOR AND TERMINAL SETS

The operators, and terminals, which are used by GP as primitives, are selected by the user based on the problem to be solved. For example to solve block stacking, the terminals used are global variables that are set to the current top of the stack, the next needed block, and the top correct block; and the operators are 'move to stack', 'move from stack', 'do until', 'eq', and 'if'.

The system is also told how many arguments each operator uses. In the above block stacking example, the number of arguments are 1, 1, 2, 1, and 3 respectively. This is to inform the system of the number of subtrees that will be placed beneath each node that is initialized to a specific argument. This information is needed when the parse tree is generated during the original population creation, and during mutation, if mutation is used.

Operators and terminals have to be designed to maintain closure of the system. In other words, operators and terminals are chosen such that all operators can take, as input, any terminal or the result of any operator (including itself). Furthermore, all error conditions that can occur in an operator given all possible input combinations, must be trapped (and handled automatically so the GP program is not halted) or the functions that are used as the operators be designed such that any errors can never occur (such operators

are called 'protected functions'). For example if the arithmetic function '+', '-', '*', and '/' are used they must be rewritten to be protected from overflow, and '/' has to be protected from division by 0.

The tailoring of primitive operators and terminals to the problem seems like cheating at first, but this is actually an unfair criticism. In all machine learning systems developed so far, problem dependent methods and terminals are used, although they are not usually explicitly discussed, or even noticed. The reason for this, as pointed out by Koza ([Koza 92] pg. 88), is that most other systems are created with a narrowly defined set of tasks in mind. When handling a specific task, no alternatives to the set of primitives are needed, and hence, by a lack of comparison, no attention is drawn to them. Koza presents a few examples for consideration:

The ID3 algorithm (Quinlan 1986) produces a decision tree that can classify an object into a class. Each object has several attributes. A certain value of each attribute is associated with each object. The ID3 algorithm constructs a decision tree that, if presented with a particular object, classifies the object into a particular class. The internal points of the decision tree consist of attribute-testing functions, which test the given object for a particular attribute. Before one can use ID3, the user must select the set of attribute-testing functions that can appear at the internal points of the decision tree. ID3 does not make this selection for the user. . .

This same determination of primitive functions and terminals occurs in heuristic systems for the induction of scientific laws from empirical data, such as BACON (Langley et al. 1987). BACON requires the user to supply a repertoire of heuristic rules (i.e., the function set) and to identify the independent variables of the problem (i.e., the terminal set) . . . For example, BACON cannot induce Kepler's Third Law from the empirical data if the user selects a repertoire of heuristic rules involving only the functions of addition and subtraction (but not the functions of multiplication or division). Similarly, it will not be possible to induce Kepler's Third Law using BACON if the empirical data provided to BACON includes the diameter of the planets but not the distance from the sun . . .

Before Jefferson, Collins et al. could begin their search for a neural network to solve the artificial ant problem [Jefferson, Collins et al. 1991] . . . they selected MOVE, RIGHT, and LEFT as the set of primitive functions for their problem. They decided that the output of the neural network at each time step would activate one of those three primitive functions. Similarly, they decided that a signal representing the presence or absence of food on the grid in the position directly in front of the ant would constitute the input to the neural network. In the field of neural networks, these steps are referred to as the process of identifying the inputs and outputs of a network. ([Koza 92] pp. 88-89)

Another possible reason that choosing primitives seems natural in other machine learning paradigms, while it seems suspicious in GP, may be the result of a natural tendency to compare GP created programs to human created programs. Programs generated by the other paradigms seem very different from those produced by humans, and so the comparison does not arise. When humans create a computer program, they use primitives that are general purposed to allow for greater flexibility in programming. But these programs are created using human intelligence. Machine learning has not yet reached this level. Genetic Programming should not be compared (at this stage anyway) to Homo Sapiens, but to other machine learning algorithms. Here GP gives an impressive performance. It appears remarkably robust; GP, with a judicious choice of primitive functions and terminals, can solve a remarkably large range of tasks; tasks that previously required completely different machine learning methods, let alone primitives, to be solved.

4.3.3 RANDOM GENERATION OF THE POPULATION OF LISP STATEMENTS

When GP is begun, a random population needs to be created before the main loop is entered. To generate a LISP expression from the set of functions F , and the set of terminals T , one uses a recursively defined algorithm:

- randomly select a value for the current node of the parse tree from set F or set T ;
- if $t \in T$ is chosen, it is a terminating value for the branch it is on
- if $f \in F$ is selected, then the process is recursively applied for the $k(f)$ branches, where $k(f)$ is the number of parameters f takes.

There are two methods for generating the initial population. The first is called the "full" method. If the level of the current node is less than a maximum tree height, the value

For the node is selected from the function set. If the node's level is equal to the maximum tree height, the node is selected from the terminal set, completing the branch. The maximum tree height for tree generation must be less than or equal to the general tree height restriction of the algorithm as a whole (the tree height restriction will be discussed in section 3.3.4), and is usually much less than it. The second technique is the "grow" method. Here the tree is generated dynamically, by randomly selecting the value of the node from the union of the F and T sets, if the node's level is less than the maximum tree height for tree generation. If it is equal, the node's value is selected from the terminal set only. The "full" method produces full trees, while the "grow" method produces randomly shaped trees. A hybrid technique called the "ramped half and half" is used by Koza. An equal amount of trees are to be produced with the tree heights from 2 to the maximum tree height for tree generation. When generating trees for a given tree height, half are produced by the "grow" method and half by the "full" method. This produces a wide variety of trees with different heights and shapes.

4.3.4 REPRODUCTION

The main reproductive techniques used are the reproduction and crossover operators. The reproduction operator makes a copy of a selected chromosome for insertion into the new population. Reproduction is used during the generation of a new population with a probability P_r , which is chosen by the user. The total probability of selecting the various genetic operators, during the generation of the population, must sum to 1. So if the probability of crossover being applied is P_c , then P_r must equal $1 - P_c$. The selection of the chromosome used for any genetic operator must be based on its fitness. The selection methods used in GP are the same as those used in GA: fitness proportional selection, rank

selection, and tournament selection. The behavior of the selection methods is also the same for both systems.

The crossover operator used in GP works through the exchange of subtrees, as described in section 4.2.2. The crossover points in the parents are chosen independently of each other, but their selection is weighted to favor internal nodes. This allows larger structures to be crossed, instead of constantly interchanging 2 terminals. When this happens, more diverse program structures are produced, increasing the exploration done in successive generations. In all experiments done by Koza the ratio for the selection of crossover points was 90%, for internal nodes, to 10% for terminals.

There is also a maximum height restriction imposed on the offspring of the crossover. If, as a result of a crossover, an offspring has a tree height that exceeds the maximum, it is rejected. To replace the rejected offspring in the new population, one of its parents is randomly selected, and reproduced, taking its place. Koza claims that the height restriction is to prevent unusually large trees from consuming an undue proportion of the computation time of the system. If a parallel implementation of the algorithm were used, he predicts, the restriction would become unnecessary. However, the height restriction has, potentially, other effects on the system. It fosters a selection pressure toward bushier trees, which may have an effect on the rate of convergence. It also limits the size of the search space; with an unlimited search space it is unclear if GP would work at all. A test between two parallel implementations, one with a tree height restriction and one without, would be needed to decide which analysis is correct.

4.3.5 THE FITNESS FUNCTION

As in GA, fitness is the property of an organism which is quantized, and is to be maximized (or, equivalently, minimized). Unlike GA, however, instead of the organism representing a single solution to a problem, the organism represents a potential program to solve the problem in general. The major technique to evaluate a program, and produce a fitness value, is to try it on a suite of test cases, usually with known solutions. In other words, the program is evaluated on a subset of its possible inputs.

There are many different measures that can be used to produce a fitness value from the test cases. If the program produces a numerical result, it can be compared with the expected result, and the errors will be summed over the test cases. If the program has a Boolean output, success or failure, then the number of successful test cases can be used as the fitness value. If the program returns a value that is to be maximized, the sum of those values would be the resulting fitness value. There are of course many more fitness measures that are used by the different GP applications; probably as many as there are applications themselves.

The above fitness measure is called the raw fitness in GP. There are two other types of fitness measures used in GP: standardized fitness, and adjusted fitness. Standardized fitness is a mapping of raw fitness to values that have the following properties: 0 is optimal, all values are positive, and a higher value implies a poorer fitness level. Adjusted fitness is a function of standardized fitness:

$$a(C) = \frac{1}{1 + s(C)}$$

where C is a chromosome being evaluated, and $a(C)$ and $s(C)$ are the adjusted and standardized fitness respectively. The adjusted fitness is used for Fitness Proportional

Selection, which requires non-negative fitness values that increase as fitness increases. The adjusted fitness fulfills those requirements.

The adjusted fitness has another property advantageous to Fitness Proportional Selection: as the standardized fitness approaches 0, the differences between the produced adjusted fitness values gets exaggerated. This is a useful property. At first a GA only needs gross differences to improve the population, however as the population's fitness improves, the smaller differences become more important. This is just the behavior produced by adjusted fitness². Adjusted fitness is not used with rank or tournament selection.

4.4 THE DIFFERENCES BETWEEN GENETIC PROGRAMMING AND GENETIC ALGORITHMS

Genetic Programming is closely based on the Genetic Algorithm. However, there are quite a few areas where GP diverges from the standard techniques and structures used in a GA system.

The first obvious difference is the use of a non-standard chromosome in GP. In GA a chromosome has a fixed length and a static, linear, structure. The chromosome used in GP differs on all points: it is non-linear, being a tree; and each of the trees can be of different shapes, so neither the structure nor the length is fixed.

² Adjusted fitness does have a downside, when used in a regular GA. An unusually good solution would have an even larger fitness value, in comparison with the other members, and could swamp the population with copies of itself. This would lead to premature convergence. However, premature convergence is not a problem in GP (see section 4.5).

Since the structure is different from chromosome to chromosome, comparing loci between chromosomes becomes meaningless. Therefore the definition of an allele (an alternative gene to the one expressed at a locus) has to be redefined, or abandoned.

GP also introduces a special crossover, that can manipulate the tree structures used in the chromosomes. This new crossover differs from the regular GA crossovers on many points. The first obvious difference is that the GP crossover works on subtrees, as an indivisible group, instead of individual loci. While it is only the uniform crossover that affects individual loci, rather than groups of loci, all crossovers can be thought of as using a template which determines whether the genes, at a given locus, will be exchanged [Syswerda 89]. No two loci can be found that must be exchanged together. However, with the crossover in GP, internal nodes can only be exchanged if the subtree they are within is selected; they cannot be individually accessed. A second point of difference is the bias that the crossover operator uses in the selection of crossover points (90% selection of an internal node, 10% of a terminal node). However, in the GA literature, there are many applications that use specialized crossovers with biases, so GP is not unique in that respect.

The difference in behavior, between the standard GA crossovers and the GP crossover, is most evident when both parents selected have the same genotype. This is called an incestuous crossover. When using a standard GA crossover, the offspring produced by incestuous crossover have the same genotype as their parents. However, when using the GP crossover, each parent has a different crossover point selected. Consequently, the offspring will be very different from the parents, even having different heights.

Another difference between GA and GP involves the use of the mutation operator; GA uses it, and GP does not. While it has been experimentally shown that the mutation operator is not needed in GP, it would be interesting to understand why it can be dispensed with. Koza notes that mutation in a regular GA is to replace alleles that have been

prematurely bred out of at given locus, but dismisses it as an unimportant secondary operation.

4.5 NICHES AND PROBABILITY OF CONVERGENCE

GP, like a regular GA, is not guaranteed to find a solution during any particular run. Although an exact solution is not necessarily required (although it is often found) and some level of tolerance for an acceptable fitness value may be given, the GP (or any GA) may still not find the correct value, even within the tolerance. This is because of the probabilistic nature of the algorithm. If, during the population creation, the system is given a set of solutions that are "far" from the goal (as measured in the number of crossovers necessary to reach the goal), or if it is unlucky in the organisms chosen by the selection routines, or the crossover points that were chosen within the organisms' chromosomes, then the experiment may not converge.

Another problem that could occur is premature convergence. **Premature convergence** happens when a member of the population is abnormally fit, but not optimal (in fact it may be far from optimal), and outbreeds the rest of the population becoming the dominant member. This stifles exploration, and progress in the GA comes to a halt. GA systems that use **Fitness Proportional Selection** are particularly susceptible to premature convergence. Premature convergence is an unavoidable liability for a GA since the reason causing it to be a problem is the same reason that the GA works in the first place (the allocation of exponentially increasing trials to schemata with greater fitness).

Rank and Tournament Selection tend to dissuade premature convergence. With Rank Selection the amount of premature convergence that can occur is controlled by varying the parameter 'max'. 'Max' is the expected number of offspring that the maximum

ranking member of the population will produce under Rank Selection. Decreasing 'max' reduces the chances of premature convergence (unfortunately it also reduces the rate of actual convergence as well) because it lowers the number of offspring that the largest ranking member produces, thus reducing the chance that it will become the only member around.

GP is not as susceptible to premature convergence. Crossovers of incestuous parents do not produce offspring that are identical to the parents, and since crossover is the dominant way that members are reproduced, no single individual can take over a population. However premature convergence can still occur in the form of a loss of subprogram alleles that are necessary to produce a correct solution.

Both non-convergence and premature convergence are usually handled by restarting the experiment. This allows the system to solve the problem with a new population that, hopefully, does not have the liabilities the old initial population had (such as the accidental inclusion of an abnormally fit individual). A restart could use the same initial population, if it is thought that unlucky crossover points resulted in the poor performance, but this is usually not done. The original population was randomly generated, and is in no way sacrosanct.

Koza justifies the restart of the system by likening a sample population in an experiment to a niche in biological systems: "... in nature, biological niches tend to be dominated by a single species (Maguarran 1988). Premature convergence in a particular niche (and occasional complete failure in a particular niche) is the norm in nature. Thus, each particular run of a genetic algorithm can be viewed as a niche, and, particular runs of a genetic algorithm can and will become dominated by particular sub-optimal species" ([Koza 90] p. 106).

The probability of convergence for an application in GP can, unfortunately, not be predicted. It must be found experimentally by running many trial experiments. The

probability of convergence P_c can therefore be estimated by $\hat{P}_c$, which is calculated by dividing the number of successful experiments by the total number of experiments run. Since each experiment is independent of each other, $\hat{P}_c$ has a binomial distribution. The equation for estimating the probability of success, given a binomial distribution, within a plus or minus bound (this can be found in any elementary statistics text; [Mendenhall, et. al. 81] was used) is:

$$\hat{P}_c = \frac{Y}{n} \pm Z_{\alpha} \sqrt{\frac{\hat{P}_c \hat{Q}_c}{n}};$$

where Y is the number of successful runs; $\hat{Q}_c = 1 - \hat{P}_c$; n is the number of experiments performed; α is $1 - C_i$, where C_i is the confidence level that $\hat{P}_c$ is between the $\pm$ boundaries; and Z_{α} is the Z score for confidence level α . Z_{α} is equal to 1.282, 1.645, 1.960, 2.326, and 2.576 for confidences levels of 80%, 90%, 95%, 98%, and 99% respectively. For example if 18 runs converged out of 30, then $\hat{P}_c = 60\% \pm 18\%$ for a confidence level of 95%. If 62 runs converged out of 100, then $\hat{P}_c = 62\% \pm 10\%$ for the same confidence level.

If a good estimation of P_c has been determined, then the number of experiments that have to be run before a successful run is found (to a confidence level of z) can be calculated. Since the experiments follow a binomial distribution, the probability of a successful run is $z = 1 - Q_c^n$, where $Q_c = 1 - P_c$. Solving for n (and substituting back P_c) gives the equation:

$$n = \frac{\log(1 - z)}{\log(1 - P_c)}$$

For example, for a 99% chance that at least one run will converge for an application with $P_c = 37\%$, after 50 generations, n is calculated as:

$$n = \frac{\log(.01)}{\log(.73)} = 14.633.$$

Therefore, 15 experiments, with each experiment running for 50 generations, must be performed to have a 99% chance that one will converge.

4.6 APPLICATIONS USED IN THE THESIS

In "Genetic Programming: On the Programming of Computers By Means of Natural Selection" Koza demonstrates the power of the GP paradigm by testing it on a very large set of applications. These include: cart centering, the animat problem, symbolic regression, programming a Boolean multiplexor, the empirical discovery of Kepler's third law, symbolic integration and differentiation, the broom balancing problem, robotics planning, automatic programming of a randomizer, game strategies, grammar induction, block stacking, recursive sequence induction, and many others. To demonstrate how a typical GP application would be designed, three applications are presented: simple symbolic regression, programming a Boolean multiplexor, and recursive sequence induction. Those three problems shall also serve as the test suite used to experiment on the system developed for this thesis.

The three problems were chosen on the following criteria: they were computationally inexpensive, so many experiments could be done in minimal time; they were distinctly different problems that would have unique behaviors, and not just be variations on a single theme; and the probability of convergence of GP for the application had to be in the GP literature for comparison purposes (unfortunately, most problems did not have this information given). All the experiments used fitness proportional selection, and were run for 51 generations. They also had the following parameter settings: a probability of crossover of 0.9, a probability of reproduction of 0.1, a maximum tree height of 17 for crossover, maximum tree height during initialization of 6, and no elitism.

In the next three sections, the three problems will be described, and a GP implementation will be summarized. To describe the GP system used to solve each problem, the terminal set, function set, fitness function, and population size will be defined

Symbolic Regression

Regression involves fitting a set of data to a mathematical series, by varying the coefficients of the terms in the series, that minimizes some measure of error between the function produced and the original data points. For example, linear regression finds the line that best fits the data points, while Fourier "regression" fits the data to various harmonics of sine and cosine functions. Symbolic Regression both selects the best form of the function to linearize, and finds the coefficients.

In the experiment, the goal form to be discovered is $x^4 + x^3 + x^2 + x$ where x can be between $[-1, +1]$. The terminal set is $T = \{x\}$, where x is a global variable. the function set is $F = \{+, -, *, /, \sin, \cos, \exp, \log\}$ which take 2, 2, 2, 2, 1, 1, 1, and 1 arguments respectively.

The functions $+$, $-$, $*$, $/$, and $\exp$ are protected against overflow (they return the value *top* which can be set to the largest number the computer can handle), $/$ is protected against division by 0 (it is set to return the value of 1), and $\log$ is protected against values less than or equal to 0 (it returns the logarithm of the magnitude of the negative numbers, and returns 0 for an input of 0). With the above protected functions, each function produces floating point numbers, and can take any floating point number as input. Therefore the system is closed.

The population size is set to 500. The fitness cases are composed of 20 randomly selected values for x between $[-1, +1]$. The standard fitness is the sum, over the test cases, of the absolute difference between the produced s-expression and the correct answers. The

probability of convergence for symbolic regression, with the goal $x^4 + x^3 + x^2 + x$, is 35% by generation 51.

Programming a Boolean Multiplexor

A Boolean multiplexor consists of n data lines and $\log_2 n$ address lines. The multiplexor takes the binary value at the address lines where individual address lines represent the 1s and 0s of a binary number, and selects the corresponding data line for output. The 6 bit multiplexor problem is designing a function that takes 2 address lines, and 4 data lines as input values and returns the value of the data line selected by the address lines. For example if address line $A0 = 0$ and address line $A1 = 1$, then the binary number they represent is 10b or 2 in decimal form. So the second data line $D1$ is selected. If $D1 = 0$, then 0 is returned, if $D1 = 1$, then 1 is returned.

For GP to solve this problem, the terminal set $T = \{A0, A1, D0, D1, D2, D3\}$, the function set $F = \{\text{and, or, not, if}\}$, which take 2, 2, 2, and 3 arguments respectively. Since all function take Boolean arguments and produce Boolean results, and all terminals are Boolean, the closure property is maintained. The population size is set to 500. The fitness cases are the $2^6 = 64$ Boolean settings of the six input variables (this is the complete description of the function, not a subset of it as is normally encountered when creating test cases for programs). The raw fitness is the number of correct cases matched by the evaluated s-expression (randomly selecting the output would average 32 matches out of the 64 cases). The standardized fitness equals $64 - \text{raw fitness}$ (i.e., the number of incorrect values produced by the proposed solution). The probability of convergence is 67% by the 51st generation³.

³ There is a discrepancy between [Koza 90], and [Koza 92], in the probability of convergence given for programming the 6 bit multiplexor, using the same system configuration. The probability was given as 67% in [Koza 90], and 28% in [Koza 92]. The higher percentage was chosen to give GP the benefit of the doubt.

Recursive Sequence Induction on the Fibonacci Series

Recursive sequence induction finds a recursive formula that predicts the values in a sequence. The Fibonacci sequence is defined recursively as $s_j = s_{j-1} + s_{j-2}$, $s_0 = 1$, $s_1 = 1$. The first 10 values of the sequence are $s = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55$.

To find the recursive definition of the Fibonacci series, GP was given the terminal set $T = \{J, 0, 1, 2, 3\}$, where J is the sequence index (for example when $J = 3$ is passed to a correct program, s_3 is returned). The function set $F = \{+, -, *, \text{SRF}\}$. The functions '+', '-', '*' are the protected arithmetic functions, as defined for symbolic regression; they take 2 arguments each.

The function 'SRF' takes 2 parameters, k and d , where k is any index less than the current sequencing index, $J = j$, and d is the default term, if the value of $k \geq j$. The 'SRF' function returns the value of the s-expression evaluated at $J = k$ (this is the s-expression produced by GP, not the target s-expression which is the true Fibonacci definition). SRF stands for 'self referencing function', and is implemented as a look-up table, which is filled by the value returned by the s-expression after evaluation for each successive value of j . If j' is the greatest value of J that has been run using the same s-expression, the s-expression should not execute with $J = j$ when $j > j' + 1$. This is to ensure that all the table values will be filled for $J = 0, 1, \dots, j - 1$.

The fitness cases are the first 20 values of the Fibonacci sequence. Both the raw and standardized fitness is the sum, over the 20 fitness cases, of the absolute value of the difference between the evaluated s-expression at position j , and the true Fibonacci number at position j . The probability of convergence is 22% by generation 51.

4.7 A NEW SCHEMA DEFINITION: AN ATTEMPT TO EXPLAIN GP USING THE SCHEMA THEOREM

One of the consequences of the difference between the structure of the chromosome in GA and GP, is the need for a new definition of a schema. The schema, defined for a GA, uses the notion of a similarity template across the loci in the chromosome. With the removal of the notion of loci, this definition becomes meaningless. Koza introduces a new schema for GP which he claims can be used as a basis for understanding why GP works. His schema is a set of subtrees that are common across a subset of chromosomes. An example is given in Figure 4.5.

Koza points out that Fitness Proportional Selection allocates trials to any structure that is associated with highly fit chromosomes, so the analysis, that is the basis for the schema theorem, is as valid for the new definition of a schema in the GP chromosome, as it is for the original schema in the linear chromosome.

Chromosome in the Population		GP Schemata, and Which Chromosomes They Match In The Population	
	Chromosome	GP Schema	Matches Chromosome
e1	$(+ (* 1 2) (\sin x))$	$\{(\sin x), (\cos 3)\}$	e3, e4
e2	$(\sin x)$	$\{(\cos 3), (* 1 2), (\sin x)\}$	e3
e3	$(- (\cos 3) (* 1 (+ (* 1 2) (\sin x))))$	$\{(+ (* 1 2) (\sin x))\}$	e1, e3
e4	$(- (\sin x) / (- x 2) (\cos 3))$	$\{(\sin x), x\}$	e1, e2, e3, e4
		$\{(\sin x), (- 1 2)\}$	none

Figure 4.5 An Example of GP Schemata

There are a few problems with this analysis. The full schema theorem not only takes into account the selection method's affect on the number of chromosomes which match a given schema, but also accounts for the detrimental effects of the crossover on the number of copies of that schema in the population. No such analysis is presented for the effect of the GP crossover on the GP schemata. Furthermore, the "proof" that a GA improves its population is a consequence of the calculation that produced the N^3 result; no such calculation is given for GP. Finally, even if the GP schema is accepted as valid, and the schema theorem is applicable, the Schema Theorem itself is an incomplete theory for describing the behavior of the GA system. The N^3 result, which is used to indicate how many *useful* schemata are processed by a GA, is only valid near the beginning of the search ([Grefenstette & Baker 89] p.26). Furthermore the Schema Theorem does not give an understanding of how crossover or mutation helps in the search (crossover and mutation is just seen as being disruptive), therefore GP's more complicated crossover cannot be explained by invoking this theory either. In summary, an understanding of why GP is successful, and which elements of the algorithm are necessary for it to work, is not in place.

CHAPTER V: EVOLUTIONARY PROGRAMMING WITH INTRONS

5.1 KEEPING CLOSE TO GA: AN EP-I OUTLINE

The **EP-I** system (which stands for 'Evolutionary Programming with Introns'; introns shall be described next chapter) has been designed to act as a canonical GA, in all ways, and yet still solve program induction applications. This includes using the standard crossover and mutation operators, developed for GA, on the structures that encode a computer program. That this is possible is not obvious. To allow for complete manipulation of a chromosome by the GA, it must be shown that the operators manipulating the chromosome produce results that decode into valid programs. In other words, one must take arbitrary segments out of a program, and, after manipulation, be guaranteed that the result is still a valid program. As seen in chapter III, all previous attempts at solving program induction using evolutionary methods are either limited, or have variable structure chromosomes which require special genetic operators, different from those used by well studied GA systems.

GP accomplishes program induction, by swapping sub-expressions, which are guaranteed to be valid sub-programs, thus keeping the integrity of the programs that are being manipulated, which produces valid offspring. As seen before, the price of doing this was to diverge significantly from the standard GA, both, by varying the length of the gene, and by using a completely new crossover. This makes the theoretical knowledge developed for GA inapplicable to the understanding of GP. Yet, as the most successful system developed so far, it would be advantageous to use as much of the innovations developed for GP, as possible, when creating the new system.

As mentioned in chapter III there are two main ideas that were introduced by GP that were not in the earlier attempts at program induction: the use of problem specific primitives, and the use of parse trees as chromosomes. The use of problem specific primitives is easily transferable to a pure GA system, provided the system uses a symbolic alphabet (a binary alphabet can be used, by giving the primitives a binary code, but it is more useful to use the symbols explicitly). However, the use of the parse tree as a chromosome is not as easy to transfer. The very reason that Koza used the non-traditional, tree swapping approach to crossover in GP was to accommodate the non-linear tree structures of the parse tree. The traditional crossovers in GA, such as the 1-point crossover, are defined only for linear chromosomes. A naive application of the traditional crossovers would render the resulting programs of the offspring completely meaningless. In order to be compatible with the genetic operators used in a canonical GA, a chromosome must, therefore, have the following two properties:

- 1) the chromosome must be linear
- 2) after any genetic operator is applied, the resulting chromosome must produce a syntactically correct computer program.

The second property will be called **Syntactic Closure** for the rest of the paper. EP-I encodes the parse tree in a way that conforms to both of the above requirements.

The basic technique of EP-I, which allows a chromosome, called the **expanded parse tree**, to be encoded with both of the above properties intact, is presented in section 5.2, but restricted to programs with only binary operators, to allow for more simple explanations. Section 5.3 describes the encoding of the expanded parse tree into a linear chromosome. Section 5.4 extends the algorithm to allow programs with n-ary operators, and section 5.5 describes the various genetic operators, as they are applied to EP-I, and shows that both properties hold, ensuring compatibility between the chromosome encoding and the genetic operators.

5.2 A SIMPLE DESCRIPTION OF EP-I

5.2.1 THE FIXED HEIGHT EXPANDED PARSE TREE

Although it shall be generalized later in the paper, the operators used in the parse tree shall, for now, only have two parameters (i.e. they are binary operators). This simplification is used so all nodes will have equal fan-out (two). This allows the arbitrary substitution of one operator for another, without the need to worry about expanding or contracting the number of parameters. Remember that parameters can be viewed as subexpressions: either an s-expression or a terminal. If you allow the operators to have a variable number of parameters (fan-out is not constant), then, when you switch an operator for one that has fewer parameters, it is unclear what to do with the extra sub-expressions (subtrees). This is important when considering mutation. The extension of the algorithm to handle n-ary operators is given in section 5.4.

The property of syntactic closure places a necessary precondition on the formation of the chromosome: the meaning of each locus must be fixed so the crossover operator can exchange genes with their alleles, and not with incompatible genes. This naturally leads to the need for the chromosome to have a fixed structure. If the structure is not fixed, then, when the standard crossover operators are applied, they would blindly cross nodes that could be holding incompatible data types, destroying the meaning of the chromosome. When the chromosome is decoded, nonsense is produced.

As seen previously, GP uses a variable structured genotype. This is because the parse trees can be of any shape or height. Remember, however, that the parse trees that are produced by the creation and reproduction routines have a maximum tree height that they must stay within. If a parse tree is produced with a greater height than the maximum it is

rejected. This suggests a way of using parse trees as the genotype and yet still keeping a fixed structure for the chromosome. If you view the genome as a full binary tree, expanded out to its maximum depth, with the original parse tree as a subtree, then the length of the genotype is actually unchanging. Therefore the length of the chromosome is now fixed.

5.2.2 THE SELECTOR OPERATOR: EMBEDDING THE PARSE TREE

While the above observation allows one to fix the size of the chromosome, it does not necessitate a fixed structure within the chromosome. An 'obvious' place to embed the original parse tree (called, simply the parse tree here after) in the fully expanded tree is at the top of the tree (with the root of the parse tree equal to the root of the expanded tree). This leaves empty gene loci below the parse tree's terminal nodes. This is an unhappy state of affairs, since, if these empty nodes mutate, we are confronted with disconnected tree fragments. In other words, if subtrees form under the original parse tree's terminal nodes, they cannot be reached.

The problem is a natural outgrowth of ignoring the requirement for a fixed structure. One of the genes may be from the terminal set, while the other may be from the operator set, or even a NULL node. This arises because the parse trees of the parents may be of different shapes and heights.

Fortunately a simple observation leads to a slightly different embedding of the parse tree, which will solve the problem. While the terminals of the parse tree are also its leaves, this is not true for the above embedding of the parse tree in the expanded tree. It is, therefore, intuitive that the terminals should "sink" in the expanded tree, and become leaves there too. In this embedding of the parse tree, all internal nodes are operators, all leaves

are terminators, and no NULL nodes are used. Therefore when crossover is applied, each matched pair of loci will be drawn from the same set; if the pair are leaves of the expanded tree, both genes are from the terminal set; and if the pair are internal nodes, both genes are from the operator set. The internal structure of the chromosome is now static.

The embedding of the parse tree into the expanded parse tree as defined so far is obviously incomplete. If the NULL nodes are removed, the expanded tree will be mismatched with the parse tree : the number of operator nodes will usually be more numerous than the number of operators in the original parse tree; the number of leaves will usually be greater than the number of terminals in the original parse tree. To solve this problem, a small finite set of related operators, called selector operators, are added to the GP operator set. A selector operator is a simple function that selects one, and only one, of its parameter values, and returns it. When used as a node in a parse tree, the selector operator effectively selects one, and only one, outgoing edge for traversal. In a binary tree there are only two outgoing edges, so there are only two selector functions: selector-L (also called selector-0) which selects the left edge; and selector-R (also called selector-1) which selects the right edge. These operators can now be used to properly embed the parse tree into the expanded tree.

The basic idea, for the embedding, is to use the selector operators to route the traversal of the expanded tree "around" areas that do not correspond to any part of the original parse tree. Here is a more formal definition. Let an expanded tree be a full tree with a fixed height that is greater than or equal to the height of the original parse tree. The value of each node in the parse tree must appear in the expanded parse tree. These are called the 'corresponding values'. The positioning of the corresponding values in the expanded tree has the following restrictions:

- 1) If Node B is the k^{th} child of Node A (in the binary case $k \in \{0, 1\}$) in the parse tree, then the corresponding value of node B must be in the k^{th} subtree of the corresponding value of node A.
- 2) Node B must be the child of Node A, or a path must exist from Node A to Node B, such that if n_i is an internal node in the path, then, if n_{i+1} is the k^{th} child of n_i and n_j is also in the path, then the value that n_i holds is the operator selector- k . A path that has this property, minus the two end points, will be called a 'selector chain' (each node on the selector chain holds the selector operator that selects the next node on the selector chain).
- 3) The corresponding value of a terminal in the parse tree, must be a leaf in the expanded tree.
- 4) The root of the expanded tree must hold the corresponding value of the root of the parse tree or be a selector operator that is the start of a selector chain that leads to the corresponding value of the root of the parse tree.

The nodes that are not corresponding nodes, nor in a selector chain, can hold any value, as long as the value is from the correct set: leaves must have genes from the terminal set; and internal nodes must have genes from the operator set. Although they do not affect the result when the expanded parse tree is collapsed, the restriction is observed to ensure that when crossover is used, the genes that are exchanged are compatible. If these properties hold, then the parse tree is properly embedded in the expanded tree (see figure 5.1 for an example of a parse tree embedded in an expanded tree). When an expanded tree has a parse tree correctly embedded in it, it is called an **expanded parse tree**.

There is more than one embedding of a parse tree into an expanded tree (as long as the parse tree is not full, or, if full, has a smaller height than the expanded tree). The proof that this is true is simple. Since the parse tree is smaller than the expanded tree, there is at

Figure 5.1 A Parse Tree Embedded in an Expanded Parse Tree

least one path from the root to a leaf in the parse tree that is smaller than the height of the expanded tree. Since the path from root to leaf in an expanded tree must hold only selectors or terminals, or it would break either rule 2 or rule 4, the path must hold at least one selector operator. A selector operator can always be changed to another selector, with

the subtree under the new selector set to be equal to the subtree under the old selector. This creates a different expanded tree, but both are correct embeddings of the same parse tree.

5.2.3 THE COMPATIBILITY OF THE EXPANDED PARSE TREE WITH TRADITIONAL GENETIC OPERATORS

In section 5.1 it was stipulated that, in order to be compatible with traditional genetic operators, the chromosome must be linear, and must hold to syntactic closure. The linearity property must hold for a simple reason; all of the traditional genetic operators were defined with linear chromosomes in mind. The syntactic closure property is necessary for the equally obvious reason that a GA should not have to waste time dealing with syntactically incorrect offspring. Furthermore, if the genetic operators only produce syntactically incorrect offspring (unless really lucky), then convergence of the GA is impossible. There is, of course, no guarantee that, if both of the two properties are enforced, the system will converge; that can only be determined empirically. However, without the two properties, attempting program induction with GA would be a lost cause.

The expanded parse tree, introduced for the EP-I system, was designed to conform to both properties. While the expanded parse tree is non-linear, it can be written in a linear fashion. This is done by traversing the tree and storing the values of the nodes in the order of traversal. The LISP s-expression itself is an example of a linearized tree; it uses a depth-first traversal with a pre-order listing. Encoding the expanded parse tree into a chromosome is described in detail in section 5.3.

The fixed structure of the expanded parse tree goes a long way to ensure that syntactic closure is ensured. However before this is explored in detail, the full EP-I system

should be presented. Syntactic closure will be revisited in section 5.5 when the effects of the genetic operators used in the system are examined.

5.3 SYMBOLIC ALPHABETS AND DECODING

5.3.1 SYMBOLIC ALPHABETS

There has been much debate on whether a GA should use a binary alphabet, or an n-ary alphabet (often called a symbolic alphabet, since the n values of the alphabet can be represented as symbols, and vice versa). The principle of minimal alphabets states that a binary alphabet produces better results for the GA than a symbolic alphabet since more "genetic information" is explicitly available for the GA to sift through (see section 2.3). However the analysis that lead to that conclusion is under dispute [Antonisse 89] (again, see section 2.3). Both Koza's GP, and Grefenstette's Classifier System, Samuel [Grefenstette 91], use a symbolic alphabet to good advantage. While still an open question, the use of symbolic alphabets seems to be advantageous when the symbols are the primitive and indivisible building blocks of the solution.

To use a binary alphabet, the 'indivisible' symbols have to be encoded as binary strings. The strings have to be long enough to enumerate all the symbols, therefore the string size encoding a symbol must be at least the logarithm of the number of symbols in the alphabet. With this one-to-one encoding, the advantages of a binary alphabet become very questionable indeed. The effects of the two different alphabets on the dynamics of a GA are most evident when observing crossover and mutation.

At first glance mutation has the same effect in either representation: either the symbol changes explicitly to another when using a symbolic alphabet, or implicitly by

changing the binary encoding for one symbol to a different binary encoding, and hence, a different symbol. However, there are some differences in the effects of mutation on the two different strings, that may cause one to think that the behavior of the system would change. In a string encoded with explicit symbols, a symbol can be changed to any other symbol, while in string with symbols encoded in a binary alphabet, a symbol can only be changed to one that has a single bit representational difference. Since the representations of the symbols into the binary alphabet is arbitrary, limiting mutation to change one symbol to a select subset of other symbols seems pointless. Another difference is the mutation rate: since the gene length of a binary encoded symbol, with a length of n , is longer than that of a directly encoded symbol, which has a gene length of one, the chance of a binary encoded symbol mutating (by having at least one of its genes changed to an allele) is n times greater given the same mutation rate. However, to 'solve' this 'problem', all that is needed is a simple change in mutation rate, which was experimentally derived anyway, to allow optimal convergence. In summary there are only minor difference in the use of mutation between the two systems, with the use of a symbolic alphabet having a slight advantage, since it does not restrict mutation to an artificial subset of the symbols for allele selection.

The effects of crossover using the two different methods are more difficult to analyze than mutation, due to the wide variety of different crossover methods. In neither GP, nor Samuel, does a binary version of crossover make sense. Both GP and Samuel use a variable length gene, with special crossovers to match. Having the crossover work within a symbol is nonsensical in either system. In Samuel, the crossover does not break rules, let alone the symbols used in a rule. Even in systems that used a linear chromosome, with a fixed structure, there seems little reason to favour a binary alphabet over a symbolic alphabet. Crossing over inside a symbol, which a binary alphabet would allow, makes little sense, as a result of the arbitrary nature of the symbol's encoding in binary. With a 1-point or 2-point crossover, the effect, except for the end symbols, is identical in either

system. The crossover of the end symbols in the binary alphabet can be seen as simply a radical mutation of the symbol into another symbol; an effect that the stronger mutation in the symbolic alphabet system already accomplishes. In uniform crossover no possible work can be done in the binary method, since each symbol is being mutated during the crossover, destroying any advantage that crossover can bring to a system through the exchange of genetic fragments. A binary encoded symbolic system, therefore, offers no advantage over a system that uses a symbolic alphabet, and can be detrimental in some genetic operators such as uniform crossover.

5.3.2 THE ENCODING SYSTEM FOR EP-I

EP-I uses a symbolic alphabet to encode the programs as chromosomes. The alphabet is stored in the system, and contains all the symbols that can be used in a chromosome. The values of the alphabet are defined by the user, and are problem specific. Even the number of symbols vary from problem to problem. There are two alphabets for any given problem. One alphabet is for the Terminal set and one is for the Operator set. Symbols have to be unique within any alphabet, although between two different alphabets identical symbols can be used. The two alphabets are stored in the system, and are used when the chromosomes of the initial population are randomly generated, and when the mutation operator changes the value of a gene to a randomly selected allele.

There are two strategies that can be used to encode a program into a chromosome using a symbolic alphabet: the direct approach and the indirect approach. The direct approach to encoding uses the s-expression as the chromosome itself. The indirect approach uses a decoder and a lookup table that decodes the symbols in the chromosome into the actual s-expression to be evaluated. As will be seen in section 5.4 the simplicity of

the direct approach is offset by the flexibility, and generality, of the indirect approach. Most GA systems use the indirect approach of encoding, and hence use some sort of decoder before the fitness function evaluates the 'organism'. GP is an exception to this rule; it uses the s-expression as the chromosome and directly manipulates it during the reproduction phase. Since the indirect system was implemented for this thesis, the rest of the section will go into the details of the decoder and its variations. However, either system will allow an arbitrary s-expression to be encoded as a chromosome that can be manipulated by a genetic algorithm.

5.3.3 THE DECODER

The decoder in a genetic algorithm takes the raw chromosome (usually having a binary encoding, but occasionally having a symbolic encoding) and turns it into a valid input for the fitness function. The dividing line between the decoder and the fitness function is arbitrary, and indeed the decoder can be incorporated into the fitness function, thus having the fitness function act directly on the chromosome. However, usually the fitness function is designed to be easily understood by humans. This is done to allow a better understanding of what the system is trying to optimize. The decoder is therefore an operator that turns the chromosome from a form that is adapted for the manipulation by a GA, into a form that is more easily understood by a human.

This has a biological analog. The "fitness function" in biology is the natural course of life. Usually the "fitness function" is analyzed by the ability of the various organisms seen in nature to survive and reproduce. The chromosome is viewed in biology as being the repository of the information encoding an individual organism. While the chromosome can be manipulated using reproductive crossover, mutation and other genetic processes, it

is eventually decoded, during embryology, to form the organism, which is, then, evaluated by the process of Natural Selection. However, just as with Genetic Algorithms, this dividing line is also arbitrary. It begins to dissolve when one considers the cells in the organism individually. The decoding of the chromosome occurs continuously, and in parallel, with the "survival process" which is 'evaluating' it. Therefore, the decoding of a chromosome is not a distinct process from the evaluation of a chromosome.

The decoder can, therefore, be seen as analogous to embryology: they both are arbitrary segments of the complete evaluation process, yet allow analysts to isolate the major processes governing the selection choice. They accomplish this by segregating the mechanism that transfers the code to larger, more easily understood structures, such as organisms, for biology, and solutions to an optimization problem, for the GA.

In the case of EP-I, the solutions that the decoder produces, are valid LISP s-expressions. The chromosome is a sequence of symbols (with no LISP list parentheses), placed in an array or list. The symbols have to be from either the terminal alphabet, or the operator alphabet. The symbols themselves do not have to have meaning to the LISP interpreter (except as symbols) and need not have the same name as the operator or terminal that they represent. The matching of symbol to actual decoding value is contained in the decoding tables. EP-I uses two decoding tables: one for the operators, and one for the terminals. The tables may be extended to hold more information about the operator or terminal than just the symbol / LISP value pair. For example, in the case of the operator decoding table, each entry in the table may also have information about the number of parameters the operator takes (see section 5.4.1), the type expected in each parameter location, or the type returned by the operator.

Parse Tree

Chromosome

Template: Position of Children in Chromosome

Legend: Child 1

x

 Child 2

y

Figure 5.2 A Template for Finding the Position of Children in the Chromosome

In order to decode the chromosome, locus specific information is needed. The decoder has to know the locus's corresponding position in the expanded parse tree, and, since subtrees can be skipped, the position of the next locus to be decoded in the chromosome. All of this positional information can be hard coded in the decoding procedure, but a better way is to have a template that the decoder can read. A template is a structure that has the same form as the chromosome, thus the position of a locus directly maps to a similar position in the template, and contains useful information about each locus

Figure 5.3 A Breadth-First Encoding of a Parse Tree in a Chromosome

(such as the position in the expanded parse tree). Using a template adds flexibility to the system, easily allowing a change of encoding patterns if needed by different problems (or even a dynamic change of encoding, although how and why this would be done is not foreseen). It is advantageous to have, as a template, a list of the loci of the current locus's 'parse tree' children (see figure 5.2). There can also be non-positional information stored in a template, that may be needed for decoding, such as which decoding table to use (although this can also be determined by the position of the node in the tree) or parameter types required.

The obvious way to encode the parse tree into the chromosome is a direct one to one, in-order mapping from the parse tree to the chromosome. This would be a prefixed (operator in the front), depth-first encoding of the expanded parse tree into the chromosome: for example, the chromosome in figure 5.2 encodes the parse tree in this way, and if decoded, it would produce the s-expression:

$$(f (+ (R k 1) (L 2 8)) (g (h i 2) (/ 7 j))).$$

The encoding used in figure 5.2 is called a depth-first encoding, since the chromosome is created from the expanded parse tree by placing the symbols into the chromosome in the order determined by a depth-first traversal of the tree. Other encodings can be envisioned: for example, the chromosome can be produced using a breadth first encoding. The same parse tree used in figure 5.2 can be encoded in a chromosome in a breadth-first manner (see figure 5.3).

Compare figure 5.2 to figure 5.3. Notice how in the depth-first parse tree, each subtree is stored contiguously in the chromosome. This is true regardless of the level of the subtree. In the breadth-first encoding, however, all operators are kept in the front of the chromosome, while the terminals are kept at the rear. The above encodings are not the only two possible. If each locus is thought of as holding a position in the parse tree (through the decoding template), then any permutation of loci in the chromosome is a valid encoding of the parse tree. The advantages and disadvantages of the different encoding methods are unknown at this time. A prefix, depth-first encoding is used currently in the EP-I system, since subtrees, which represent subroutines, seem to be logical building blocks for encoding a program. However this is definitely an area that needs investigation.

5.4 N-ARY PARSE TREES

5.4.1 ENCODING A FULL N-ARY PARSE TREE

Until now, the EP-I system was defined for binary parse trees, using binary operators. This restriction was necessary to simplify the description of the expanded parse tree. However, with the background knowledge in place, an EP-I system with full trees with n-ary operators can be easily defined.

The first step is to "flesh out" the tree by setting the fan-out of each node of the tree to equal the maximum number of parameters needed by any operator in the Operator Set. This is possible, since the Operator Set, for any problem, is finite; therefore an operator must exist that will equal, or exceed, the number of parameters needed by any other operator in the set. Note that all operators must take a fixed number of parameters, otherwise, each node in the tree will have a variable number of subtrees below it, destroying the fixed structure property. Also, no optional or keyword parameters are allowed. Once this fan-out is found from the Operator Set, and the tree-height is known, a full tree can be created, which can be used as the template for a fixed length chromosome.

While setting the fan-out equal to the maximum number of parameters that any operator can use, fixes the structure of the expanded parse tree, there are still many operators that will take fewer parameters than the maximum. Therefore, the operators at many nodes will be given too many operands. To handle this problem, the operator can be applied to a subset of the values returned by the node's children. This method is similar to the approach taken in the use of the selector operators. The selector operators also ignore extraneous parameters, which allows the embedding of arbitrary shaped parse trees into a fully expanded parse tree.

Subtrees that return a value that is ignored by the operator of its parent node, must not be collapsed. If the subtrees encode operators that have side effects, they will be performed, even though the "program" does not include them. This includes subtrees, below selector operators, which are not selected.

There are three methods that can accomplish the appropriate embedding with the use of operators that take an arbitrary (but non-variable) number of parameters. These are described below.

The first method works through the redefinition of any operator which takes fewer parameters, allowing it to take the same number of parameters as the Operator Set

```

(defun foo1 (arg1 arg2 arg3)
  (+ arg1 (eval arg2)))

(defmacro foo2 (arg1 arg2 arg3)
  `(let ((arg-a (eval ,arg1))
        (arg-b (eval ,arg2)))
    (+ arg-a (eval arg-b))))

```

- Both 'foo1' and 'foo2', take three arguments, but only add the first two together.
→ with $x = 4$, both `(foo1 1 'x (+ x 1))` and `(foo2 1 'x (+ x 1))` return 5
- However, LISP functions evaluate all arguments before applying the operator
→ `(foo1 1 'x (incf x))` returns 6, instead of 5
note: `'(incf x)` increments the variable 'x' by 1, through a side-effect
- The macro was designed to completely ignore the third argument
→ `(foo2 1 'x (incf x))` still returns 5

Figure 5.4 Using A LISP Macro To Ignore the 3rd Argument

maximum, but ignore the extraneous ones. The chromosome can, then, be the LISP s-expression itself with no decoding needed. Therefore, no decoding tables are needed; all the information needed to evaluate the chromosome is present in the chromosome itself, provided all the symbols in the operator alphabet, have appropriate functions defined in the system environment. Just as the ignored subtrees must not be collapsed, to prevent unwanted side-effects, all sub-expressions that are in argument positions that are ignored by the operator of the s-expression, must not be evaluated. This has an added benefit of decreasing the evaluation time. In LISP, the selective evaluation of arguments can be accomplished by using a macro to define the operators that ignore their parameters, rather than using regular functions, which automatically evaluate all the arguments (see figure 5.4 for an example).

A second method works through the use of a decoding table which helps a decoder translate a chromosome into a LISP s-expression to be evaluated. The decoding table contains a set of triples: the symbol used in the chromosome that represents the operator, the corresponding operator that the symbol decodes into, and the number of parameters that

2 of/de 2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

the operator takes. The symbols used to encode the chromosome (the first element of the triple in the decoding table) are recorded in the system's operator alphabet, instead of having the operators themselves in the alphabet (the alphabet is the set of values that is used to create a chromosome, hence it has to consist of the symbols that comprise the chromosome, and not the operators that the symbols decode into). If the operator takes n parameters where n is less than the fan-out size of the expanded tree, only the first n parameters from the left are decoded, while the rest of the parameters are ignored during decoding and the corresponding sub-expressions do not appear in the final s-expression to be evaluated. The choice of decoding the first n parameters is a bit arbitrary, and is chosen for simplicity sake. Any selection of arguments equal to the number of arguments used by the operator is permissible.

The final method bundles the operator and parameter selection into the gene. In other words, each gene determines which parameters will be passed to its operator. The parameter selection would be stored as a vector containing the position of the parameters used in the argument list. Like the second method the chromosome, if set up this way, needs to be decoded. The vector length is equal to the number of parameters used by the corresponding operator; the values in the vector are the position of the children from left to right in the analogous expanded parse tree; and the order of the values is the order, from left to right that the parameters will take in the decoded parse tree. An example is presented in figure 5.5.

Note that the positional information for decoding is now explicit in the chromosome and is not inferred from the decoding table. The decoding table is still needed to translate the symbols into the actual functions used in the decoded parse tree. Also the operator alphabet used by the system to initialize the gene during mutation and chromosome creation, needs the information necessary to create correct Parameter Location Vectors.

Chromosome

if	=	1	2	1	/	2	4	12	mod	15	5	17
----	---	---	---	---	---	---	---	----	-----	----	---	----

Parameter Location Vector Template

[0]	[0]	[T]	[T]	[T]	[2]	[T]	[T]	[T]	[0]	[T]	[T]	[T]
[2]	[2]				[0]				[1]			
[1]												

Decoded
S-Expression

(if (= 1 1) (mod 15 5) (/ 12 2))
--

Legend:

[i] = ith child; $i \in \{0, 1, 2\}$
 [T] = terminal; no children
 'if' takes 3 parameters
 'mod' takes 2 parameters
 '/' takes 2 parameters

Figure 5.5 Decoding a Chromosome with Parameter Location Vectors

This information is held in the decoding table, so the decoding table can double for the system's operator alphabet.

5.4.2 THE SELECTOR OPERATORS IN A FULL N-ARY PARSE TREE

In order to allow a parse tree of any size to be embedded in a full N-ary Parse Tree, the proper selector operators must be defined. There are different selector operators used for each of the three methods. To help describe them, an Operator Set F will be used, consisting of: two operators that take four parameters, d_1^4 , and d_2^4 ; five that take three parameters, c_1^3 , c_2^3 , c_3^3 , c_4^3 , and c_5^3 ; and three that take one parameter, a_1^1 , a_2^1 , and a_3^1 .

Terminal set T consists of 4 terminals: t_1 , t_2 , t_3 , and t_4 . In each of the three expanded N-ary parse tree encoding methods, the main idea is to generalize the selector operators. Each selector operator (or one generalized selector operator) selects one of the evaluated children subtrees as its result. This is a natural extension to SelectorL and SelectorR of the binary parse tree.

For the first method a selector operator has to be created for each parameter up to the maximum number of parameters that an operator can take. For example, in the Operator Set F , the maximum number of parameters is four; in fact all operators are redefined to take the four parameters. Therefore, four selector operators are created, one for each of the four parameters. Each of the four selector operators take four parameters, like all other operators, and return the value of the parameter for which it is defined. The four new selector operators are added to the operator alphabet used by the system.

In the second method, the selector operators are associated in the decoding tables with the symbols used in the chromosome. Like in the first method, the number of selectors is equal to the maximum number of parameters that an operator can take. In the decoding tables, the entries for the selector operators are slightly different than for the other operators. Instead of having the triple (symbol, operator, number-of-parameters), it uses (symbol, '*selector*', n) where n is an integer set to the position of the parameter that the selector is choosing. The decoder, then, instead of returning the appropriate selector operator with all of its parameters expanded as LISP s-expressions, expands the parameter that the selector chooses, and returns that code alone, without the code for the other parameters, or the selector operator which would have surrounded them. In other words, the decoder uses the selector symbols to decode only the appropriate sub-expressions. The selector operator chromosome symbols must exist in the operator alphabet.

Two examples are shown in figures 5.7a, and 5.7b. They both use the Terminal Set T , and Operator Set F . Four triplets are added to the operator decoding table (each symbol from the triple must be unique, and exist in the alphabet):

(S0 *selector* 0);
 (S1 *selector* 1);
 (S2 *selector* 2);
 (S3 *selector* 3);

Notice that the s-expression does not contain any of the code represented by the ignored subtrees. Therefore the non-selected subtrees will not be evaluated with the s-expression, so no extraneous side-effects can occur.

Chromosome

Corresponding Parse Tree

Decoded S-Expression

$(c_2^3 t_4 t_3 t_2)$

Figure 5.6a Decoding a Chromosome Using Selection Method 2

Chromosome **Corresponding
Parse Tree** **Decoded
S-Expression**

$$(c^3_2 (a^1_1 t_1) t_3 (d^4_2 t_1 t_2 t_3 t_4))$$
Figure 5.6b Decoding a Chromosome Using Selection Method 2: A Second Example

The third method, unlike the first two methods, only requires one selector operator. This selector operator has a corresponding Parameter Location Vector of length 1, and can be set to an integer equal to the position of the parameter that is to be chosen. To demonstrate, Figures 5.7a, and 5.7b use the same examples in Figures 5.6a, and 5.6b, but with the symbol s_0 replacing all the selector operators.

As in all the methods, the corresponding symbol of the selector operator must be added to the alphabet. The selector operator can be viewed as an identity function of one parameter. It can be represented that way explicitly in the s-expression, or the selector

Chromosome

Parameter Location Vector Template

Decoded
S-Expression

Legend:

- [i] = *i*th child;
 i ∈ {0, 1, 2, 3}
- [T] = terminal;
 no children

Figure 5.7a Decoding a Chromosome Using Selection Method 3

Chromosome **Parameter Location Vector Template** **Decoded
S-Expression** **Legend:**

- $[l]$ = l th child;
 $l \in \{0, 1, 2, 3\}$
- $[T]$ = terminal;
 no children

Figure 5.7b Decoding a Chromosome Using Selection Method 3: A Second Example

operator can be ignored and only the sub-expression chosen by the selector operator will appear in the final s-expression (as in Figures 5.6a, and 5.6b).

5.4.3 COMPARISON OF THE THREE ENCODING METHODS

Each of the three encoding methods has advantages and disadvantages. The first method (having the s-expression actually be the chromosome) is direct, and needs no decoding to be evaluated. However, it has some representational limitations. It forces the user into a pre-order depth-first encoding, allowing no other encoding schemes, nor does it allow for any other manipulation during decoding that one might want to use, such as 'inversion'. During crossover and mutation it behaves identically to method 2 thus incurring no extra advantages there. The advantage in the time saved during decoding is not that significant, since most of the time is spent in fitness evaluation, and not decoding. Finally this method of encoding is less satisfying than the other two, since the chromosome is built with nested linked lists, as opposed to using an array of symbols. A nested linked list chromosome demands special copying, crossover, and mutation operators instead of using generic ones that could be provided by a general symbolic GA. As a result of its limitations, similarity in normal use to the second method, and a lack of overwhelming advantages in speed, this method was not implemented.

The second method (decoding the chromosome into an s-expression, but assuming a constant ordering of the parameters) uses an encoding that is compatible with a generic symbolic GA, and does not need specialized genetic operators to manipulate it. It is simple to implement, and requires less space than the third method. The chromosome can be encoded in a variety of ways, which allows flexibility in experimenting with different genetic encoding / manipulation schemes. The only possible downside, is the unknown

effect of assuming the left to right decoding of the subtrees, which keeps expressed genes more rigidly fixed in place than does method three. This is the encoding method currently implemented.

The third method (including the ordering of the parameters in the gene) is the most general and flexible of the three systems. It allows for any encoding technique, and any pre-decoding manipulation, or special genetic operators, such as 'inversion'. However, it uses much more memory to store a chromosome than the other two methods, and so its use may be problematic if memory is scarce. It is also slightly more cumbersome to implement, since proper Parameter Location Vectors must be formed during creation and mutation. Since it is unknown if the added flexibility of this method is advantageous, and the implementation is more time consuming, this method has not been implemented at this time.

5.4.4 SIZE OF THE SEARCH SPACE: THE EFFECTS OF A SYMBOLIC GENE

For a symbolic alphabet with a cardinality of k used in a chromosome of length l , there are k^l possible chromosomes and $(k + 1)^l$ possible schemata¹. For two alphabets, the terminal alphabet and the operator alphabet, with respective cardinalities m and n , the number of possible chromosomes depends on how many terminals and how many operators there are in the expanded parse tree. If there are $\#t$ terminals, and $\#o$ operators, in the tree, then there are $m^{\#t} + n^{\#o}$ number of possible chromosomes.

¹ The schema analysis for the symbolic alphabet is the usual analysis as presented by Goldberg, not the modified analysis proposed by Antonisse (see section 2.3).

There is usually many more terminals in the expanded parse tree, than there are operators. This can be easily proved. Let f be the fan-out of the expanded parse tree, and h be its height. Each level of the tree is growing exponentially, so $\#t = f^h$. Since the number of operators equals the sum of all the nodes from all the levels, ignoring the terminal level, then, the number of operators is:

$$\#o = \sum_{i=0}^{h-1} f^i$$

This is a sum of a geometric series, which has a closed form expression. Therefore the number of operators can be written as:

$$\#o = \frac{(f^h - 1)}{(f - 1)} = \frac{(\#t - 1)}{(f - 1)}.$$

This means there are always approximately f times more terminals than operators (for example, if the fan out is 4, there is approximately 3 times more terminals than operators).

With the introduction of the f selector operators into the operator alphabet, the number of possible chromosomes becomes: $m^{\#t} + (n + f)^{\#o}$. The number of schemata has grown to

$$(m + 1)^{\#t} + (n + f + 1)^{\#o}.$$

This is a much larger search space than for the binary alphabet where there are 2^l possible chromosomes and 3^l possible schemata. The search space grows polynomially with the number of symbols, and grows exponentially when the chromosome is lengthened. This leads to a large increase in the search space in comparison to the number of symbols added. A parameter that has an even larger effect on the size of the search space is the height of the expanded parse tree. The size of the chromosome grows exponentially with the height of the tree, and the size of the search space grows exponentially with the size of the chromosome. From this analysis, one would expect a rapid reduction in the number of

correct solutions found when the tree height is increased² (after a threshold, where the tree heights are too small to encode the solution, is passed). This experiment was not yet performed for lack of time, and is consigned for future work.

5.5 THE BEHAVIOR OF GENETIC OPERATORS IN EP-I

5.5.1 SYNTACTIC CLOSURE REVISITED

EP-I has been designed to work with the traditional genetic operators used in GA (see section 5.1). To allow for this, two restrictions were placed on the formation of the chromosome: linearity in form, and syntactic closure. Section 5.2.3 showed that EP-I uses linear chromosomes, and that the structure of the chromosome is fixed, which is a necessary condition for syntactic closure to hold. Now that the full algorithm has been presented, it is time to re-examine the effects that the genetic operators have on the EP-I system, and determine whether syntactic closure really does hold.

5.5.2 MUTATION IN EP-I

Point mutation in EP-I is a direct analog of the regular mutation that is used in GA systems with binary alphabets. Mutation occurs during chromosome reproduction, both during crossover and direct copying. As presented earlier, the mutation operator takes each gene in the chromosome as input, and randomly decides whether to copy the gene through,

² This effect may be mitigated by the masking effect of the introns (see section 6.2.3)

or change it to one of its alleles. For a GA that uses a binary alphabet the change to an allele is simply accomplished by 'flipping the bit'. With a symbolic alphabet the allele must be randomly chosen from the alphabet used to encode the chromosome. This means that when mutating a gene, the mutation operator must use positional information to determine whether to choose from the terminal alphabet or the operator alphabet. Therefore the positional template must be passed to the mutation operator as well as the two alphabets.

If for some reason one wanted to restrict the access of the template from the mutation operator to prove some theoretical point, the terminal and operator alphabets could be merged into one single alphabet, with the decoding tables supplying the appropriate interpretations during decoding; each symbol will have a dual interpretation, one as a terminal and one as an operator. If there are more terminals than operators (or visa versa) when using only a single alphabet, the extra symbols, which do not map to any operator (terminal), should be assigned duplicates of operators (terminals) that have already been assigned to other symbols. Unfortunately, this might lead to the over representation of the duplicated operator (terminal) in the chromosome during initialization, and a greater probability of insertion during mutation, since it can be selected twice as often as a non-duplicated operator (terminal). Therefore, in practice, it is better to use two separate alphabets.

The effects of a mutation depend on the encoding method used. As described in section 5.4.2, there are three methods to encode a chromosome: the direct use of the s-expression as the chromosome; the use of a decoder to turn a chromosome, that uses purely symbolic genes, into an s-expression; and the inclusion, with the symbolic genes, of a vector which holds the location of the subtrees that will be expressed.

If the first two methods are used, a mutation changes either the value of a single terminal, or operator. If the mutation occurs at an operator locus, and the operator that is being replaced takes a different number of parameters than the new operator, then some of

the subtrees, which were not selected by the previous operator, may now be decoded, and some which were selected, may now remain untouched. However, in all cases the mutation will result in a viable chromosome (i.e., one that can be decoded into a valid s-expression). So syntactic closure is preserved by mutation.

An example of the effects of mutation is presented in Figure 5.8. Remember, in the

Figure 5.8 The Effect of Mutation on Decoding

first two encoding methods for the chromosomes, the subtrees are decoded from left to right until the number of parameters are filled; the rest of the subtrees are not decoded.

If the third encoding method is used, the mutation operator has to create a Parameter Location Vector that is appropriate to the interpretation of the symbol that has been chosen. Therefore, both the decoding tables, and the positional template, are used by the mutation operator for this method. The vector is created with a size equal to the number of parameters needed by the operator (the Parameter Location Vector is not used for terminal loci) as recorded in the decoding table (this information may also be kept in the alphabet itself). The subtrees that are to be selected are randomly chosen, without replacement, from the children of the mutating node.

Instead of randomly creating a Parameter Location Vector each time a mutation occurs, the previous Parameter Location Vector can be used as the basis for the new vector. There are three possible scenarios: the new operator takes fewer parameters than the old; the new operator takes more parameters than the old; both the new and the old operator use the same number of parameters. If the new operator takes fewer parameters, then the old vector is too large. When this occurs, random subtrees are selected from the old vector, and dropped. The number of subtrees to be dropped is equal to the difference between the number of parameters taken by the old and new operators. If the new operator takes more parameters, then the old vector is too small. For this case, a random set of subtrees is chosen from the children of the node, that are not yet listed in the Parameter Location Vector. These subtrees are added, and can be either placed at the end of the vector, or randomly distributed throughout. This allows as many subtrees that were expressed in the chromosome before the mutation to be expressed afterwards.

Keeping as much of the old Parameter Location Vector as possible is desirable over randomly generating a new vector. It reduces the very large structural change to the decoded s-expression caused by the wild deletion, and insertion of subtrees that result from

random vector creation. These large changes would drive the system into instability. Mutation could no longer fine tune the genome, since it would always produce catastrophic changes.

Finally, as long as a proper Parameter Location Vector is created, a mutation operator that replaces a gene with an allele from the appropriate alphabet, will, like the first two encoding methods, always produce a new chromosome that can be decoded into a valid s-expression. In other words syntactic closure is preserved.

5.5.3 CROSSOVER IN EP-I

Syntactic closure holds under the operation of crossover because of the fixed structure of the expanded parse tree. Each locus, with the same position in the chromosome, has the same location in the expanded parse tree. Therefore, genes in different chromosomes, which have the same position within the chromosome, are still in the same position in the expanded parse tree when exchanged. Operators will only be exchanged with other operators, and terminal with other terminals. Now, any operator can be placed in an internal node of the expanded parse tree, and any terminal can be placed in a leaf of the expanded parse tree, and still result in a valid parse tree. This is a result of closure (functional closure as defined for GP, not syntactic closure); any operator must be able to take as input the result of any other operator, or any terminal. As long as closure holds, the corresponding chromosome will decode into a valid LISP s-expression. Therefore syntactic closure is maintained when crossover is applied.

Since this is true for any crossover that does not change the order of the genes within the chromosome, which is true of all fixed length GA crossovers, any crossover developed for GA can be used by EP-I. This includes all n-point crossovers such as the 1-

point and 2-point crossover, and the uniform crossover. For an example crossover, the 1-point crossover, see Figures 5.9 a-c.

A crossover operator may have different effects on the EP-I system, depending on which chromosome encoding method was implemented. The advantages and disadvantages of the various pairings are, at this time, unknown. The depth-first encoding

Figure 5.9a Chromosome 1 to be Crossed
Crossover Point is at Locus 21

Figure 5.9b Chromosome 2 to be Crossed
Crossover Point is at Locus 21

Child 1

Child 2

Expanded Parse Tree for Child 1
(Tree for Child 2 Analogous)

$$(f \quad (+ \quad (^{\circ} \quad j \quad 1) \quad 4) \\ (+ \quad (/ \quad 4 \quad (+ \quad i \quad j)) \quad 7))$$

Decoded S-Expression For Child 2

$$\begin{aligned} & (^* (+ 1 (- 1 (/ (h \mid 8) \\ & \qquad \qquad \qquad (^* j \mid k)))))) \\ & (g (h \mid (^* (+ i \mid 2) 2)) \\ & \qquad \qquad \qquad (/ 4 \mid k)))) \end{aligned}$$

Figure 5.9c Children after the Crossover, With an Example Expanded Parse Tree

seems to be a natural encoding method, and is mandatory for the crossover used by GP. The 1-point crossover would seem to be the natural pairing with a depth-first search, since the exchange of contiguous genes is, like in GP, the exchange of subtrees in the corresponding parse tree (see Figure 5.10). Certainly a random encoding would not seem to be sensible for the 1-point crossover, since there are no obvious short defining length building blocks that can be identified. However uniform crossover does not use short defining length building blocks, and so may behave identically no matter which encoding were used. All of the experiments done to date have used depth-first encoded chromosomes, because they are the most intuitive encoding (virtually a direct mapping of the parse tree into the chromosome), and have definable building blocks for the GA to act on.

5.5.4 THE COUPLED 2-POINT Crossover

All of the crossovers used for the experiments in this thesis are unmodified traditional crossovers, used by many other GA systems, for many other applications. This is, of course, done deliberately to isolate which factors are necessary for program induction to occur through evolution. However there is a natural modification of the 2-point crossover that bears commenting on. It is very reminiscent of the GP crossover. This is the coupled 2-point crossover.

Instead of just randomly picking two points, as is done in the regular 2-point crossover, only the first point is picked at random. The node that corresponds to this point is the root of a subtree (if it is a terminal, it is the root of a subtree of size zero). The second point is chosen at a locus which ensures that all genes being swapped belong to the same subtree, and no gene that belongs to the subtree is not exchanged. We have

Figure 5.10 Subtrees Which are Crossed Over, are Embedded at Contiguous Loci

now restricted the 2-point crossover to swapping one subtree with another, just as is done in GP. However, unlike GP, these are not two arbitrary trees, but trees that have the same position in the expanded parse tree.

There does not seem to be any advantage to the coupled 2-point crossover in comparison to the regular 1-point or 2-point crossover. Both the 1-point and 2-point crossovers also swap subtrees that have the same position in the expanded parse tree. However, unlike the coupled 2-point crossover, they exchange more than one tree at one time. This allows for greater cross fertilization between the two chromosomes, and so explores the solution space more rapidly (no actual experiments have been done with the 2-point coupled crossover yet).

CHAPTER VI: NOVEL BIOLOGICAL ANALOGIES AND EXTENSIONS TO EP-I

6.1 INTRODUCTION

Many of the insights that inspired the creation of genetic algorithms were spawned by biological analogies. The terminology used in the GA literature is rife with stolen words and phrases from the genetic and biological lexica. The novel features that are introduced in EP-I can also be seen to have close biological analogies. These analogies provide insights into the processes behind the new features in EP-I, and, in turn, inspire new ideas to extend the EP-I algorithm.

The use of 'introns' and 'exons' in the EP-I system is examined in section 6.2. The concept of 'gene jumping' is introduced to the system in section 6.3. Finally, an extension, which provides a simulation of the GP algorithm within the EP-I framework, is presented in section 6.4.

6.2 INTRONS AND EXONS

6.2.1 THE DEFINITIONS OF INTRONS AND EXONS

Both 'intron' and 'exon' are terms that are borrowed from Genetics. In this thesis, an intron shall mean any genetic material that does not have a phenotypic expression; and an exon, similarly, shall mean any coded genetic material that does have phenotypic behavior. The definition of exons is completely analogous to the biological definition:

exons, in genetics, are coded genetic material that are used to build proteins. The definition of the intron as expressed above, however, is not precisely the common usage in biology. The above definition of 'intron' is more akin to the biological definition of 'junk gene': genetic material that does not get expressed into proteins. An intron, in biology, is a type of junk gene that is found inside a cistron¹. Junk genes found between cistrons are called spacer DNA. The reason for the slight redefinition of the intron, is that a cistron does not have an exact analog outside of biology, so the distinction between spacer DNA and introns is not meaningful. Furthermore, since 'intron' seems a natural counterpart to 'exon', the term 'intron', as used here, is given the broader definition of the junk gene.

Introns and exons are found in eukaryotic cells (cells with a nucleus), and almost never found in prokaryotic cells (cells without a nucleus, usually bacteria). Both the introns and the exons are usually transcribed into messenger RNA (mRNA), however, the introns are removed by a preprocessing step while still inside the nucleus. The exons are spliced together, and the mRNA is transported to a ribosome, where the protein is created. Since the introns had been removed from the mRNA before it reached the ribosome, the code in the introns is not expressed as protein; therefore, there is no phenotypic effect from the introns.

The genotypic effects, and 'purpose' for the existence of introns, is still an open question in genetics. It has been proposed that they may exist because they have properties that make them good at surviving within the environment of the chromosome

¹ A cistron is a sequence of genetic material, beginning with a 'start' codon, and finishing with an 'end' codon, that encodes a complete protein [Dawkins 82]. A codon is a base pair triple (three base pairs) that encodes the basic instructions used for the creation of protein by the cell. Most of the instructions are decoded at the ribosome (the organelles in the cell's cytoplasm that produce proteins) and correspond to amino-acids, which are the building blocks used to form a protein. Some of the instruction codes, however, are used when producing the strand of RNA that is to be read at the ribosome, such as the start and stop codes.

(the nucleus), but are neither beneficial nor detrimental to the organism itself, since the effects of the introns are not felt outside of the nucleus. For this reason, junk genes have sometimes been called 'selfish DNA'; which is an analogy to the thesis behind Richard Dawkins' book "The Selfish Gene" [Dawkins 76]. Here genes are viewed as having 'selfish' behavior in their environment; and exist solely for their own propagation, not for the express benefit of the organism which they are in (although what benefits one probably benefits the other). Similarly 'selfish DNA' exists solely for its own benefit, and not for the gene it is sitting in.

For more information about introns, and exons, see [Crick 79], and for more information about DNA, RNA, and protein synthesis in general, see [Stansfield 91].

6.2.2 INTRONS IN EP-I

The notion of introns is naturally applicable to EP-I. Like eukaryotic genomes, EP-I has genetic material inside the chromosome that is not expressed when decoded. Subtrees under edges that are not selected by a selector operator, and subtrees under a node set to an operator that does not use all of its parameters, are not included in the decoded s-expression, and so do not have an effect when that s-expression is evaluated by the fitness function. It is natural, therefore, to call these subtrees introns, and all the other nodes, which are expressed in the s-expression, exons. In other words, introns can be thought of as non-selected subtrees in the expanded parse tree. In the actual chromosome used, except in a depth first encoding, the subtrees do not have to be encoded at contiguous locations, so introns can be dispersed throughout the chromosome. For an example of introns, and exons in a depth first encoded chromosome, see figure 6.1.

Introns hide genetic code from the selection process. This is both a blessing and a curse. On the negative side, this can be seen as slowing down convergence, since the selection pressures, that drive the system, are not applicable to the hidden code. Such code will not be bred out (except accidentally), even if it is useless or detrimental when expressed, nor recognized and promoted, even if it is beneficial. It is not even a good depository for safekeeping 'good' code; when hidden inside an intron, any schema can still be destroyed by the crossover operator (or the mutation operator if it is allowed to work within introns), which cannot be prevented from selecting crossover points within an intron, since the other chromosome being crossed may have an exon at that locus.

In spite of the above argument, the introns may actually help the search process. An intron can be viewed as an analog of the 'don't care' symbol explicitly expressed in the chromosome. In this respect it is comparable to the Classifier System's '#' symbol, except that introns affect more than one gene at a time. The intron dynamically blocks out segments of the genotype that are not needed for the ultimate solution. Since this reduces the amount of search space, the introns may actually speed up the convergence process. The positioning of the intron is influenced by selection pressure, and so introns can be seen as a large scale 'gene' that is expressed by blocking out unneeded segments of the genotype.

When an intron is turned into an exon, through mutation or crossover, the intronal material that is activated is frequently the root of a huge subtree, all of which, suddenly, is expressed / hidden (see Figures 5.7, 5.8, 6.2, and 6.3 for examples). This means mutation, when introns are used in the chromosome, can have a larger effect on the resulting phenotype than in most GA systems. The effects of these large sporadic mutations are unknown. It may create instability in the system, preventing any convergence from occurring. It is equally possible that the system may handle the

mutations, if infrequent enough, in stride, with the bad mutations being bred out, and the good mutations being selected and propagated.

In summary, introns may either slow convergence, by preventing selection pressure from sifting through schemata which are hiding in the introns, or increase it, by masking unneeded genetic code, reducing the search space. They may either create instability in the system through wild mutation effects, or have no effect on stability what-so-ever. Which of these disparate analyses are correct, or whether there is a complex dynamic between the various opposing effects, cannot be determined without careful experimentation. This is outside the scope of this thesis, since, before the effects of introns are investigated, it is mandatory that EP-I be shown to work. Otherwise the inclusion of introns is a dead-end concept, and not worth exploring. The positive results reported below are encouraging, and the exploration of the effects of the intron will provide an interesting topic for future work.

6.2.4 INTRONS AND THE MUTATION OPERATOR

Mutation can have a large effect on the placement of introns and exons in the chromosome. A mutation inside an intronal subtree has no visible difference. A mutation at a locus which is an exon, where both the gene and the allele take the same number of parameters (and has the same Parameter Location Vector, if the third encoding method for chromosomes is used), also has no effect on the intron/exon distribution. All introns remain introns; all exons remain exons. However, if a gene that is at a locus

Expanded Parse Tree
of The Chromosome

Figure 6.2 The Effects of Mutation on Introns and Exons

which is an exon, is changed to an allele which accepts a different number of parameters, or is a selector operator that is selecting a different subtree, then intron subtrees that were previously not selected, or were outside the range of the parameter list, could suddenly become accepted as a parameter of the new operator, or be selected by the new selector operator (see Figures 5.7 and 6.2 for some examples). Similarly exon subtrees could become introns by falling outside the range of the parameter list of the new operator, or not be selected by the new selector operator. The number of subtrees that can be affected by the mutation is less than or equal to the fan-out of a node, since these are the subtrees that are affected by the change of a single node.

The inclusion of introns into the chromosome does not affect the rate of mutation, even though some mutations are occurring inside the intron. This is because the mutation rate is the probability that at any given locus, a gene will mutate. This probability is independent for each locus, so the probability that any given exon will mutate is not affected by the mutations that occur in the introns.

A variation of the EP-I algorithm can be used to prevent the destruction of potentially good genes, which are hidden inside of an intron, by mutation. During decoding of the chromosome, all loci are labeled as either an intron or an exon (this would have to be done as a separate pass, if the chromosome is the actual s-expression, since the decoding step is skipped). Then, during reproduction, mutations are not allowed to occur at a locus that has been labeled as an intron. This will not affect the mutation rate, since, as noted before, the mutation of any locus is independent of any other locus. The number of mutations, in total, will decrease, but the number of mutations that have an effect on the system remains the same.

While this enhancement has been implemented in the system, it is not currently used. Any schema that is needed by the system, and has already demonstrated its good effects, will exist in other chromosomes in the population. In some of those chromosomes, the schema will not be an intron, since it has to have been expressed in the past to be declared beneficial, and therefore it does not need special protection. However the schema may not be present in other chromosomes because it is new and untested, or has poor fitness through incompatibility with the other genes in the chromosome, or through an epistatic relation with genes that have not been developed but which are vital to the schema for it to have a positive phenotypic effect. When these cases occur, it is frustrating to an omniscient observer, who would want to prevent their disappearance from the gene pool. However, while regrettable, the destruction of these types of schemata is not that unusual in a probabilistic system like the GA. They might, equally

as well, not be selected by the selection operator, or be destroyed by crossover. A schema that exists in only one chromosome, and has not yet shown itself to be useful, is regarded by the system as having as much chance to be detrimental, as to be productive. Since there are many other possibly interesting schemata out in the gene pool, the destruction of a randomly chosen schema will not change the likelihood of convergence.

6.2.5 INTRONS AND THE CROSSOVER OPERATOR

The effects that crossover operators have on introns depend, not only on which crossover operator is used, but also on how the chromosome is encoded. A 1-point crossover on a breadth first encoding has different effects than it does on a depth first encoding. Consequently the effects on the introns in the chromosome must be different also. Since depth first encoding is uniformly used throughout the thesis, it will be the only encoding examined here as well.

The effects that one point crossover has on introns is similar to the effect that mutation has, except more subtrees are involved. This is because the crossover breaks links between parents and subtrees from the level of the crossover point, which can be at the bottom of the tree if a terminal is selected, all the way up to the root of the tree² (see Figure 5.9 for an example). Each link that is broken has the potential to change the intronal status of a subtree. If the subtree was an intron in parent 1, when moved to

² This is true unless the crossover point is located in the subtree belonging to the last child of the root. In this case the all the links between the root and its children are preserved. This is true recursively for all subtrees throughout the expanded parse tree. This property can be generalized. For any subtree within an expanded parse tree of out-degree of k , if the crossover point is below the root's i th child, then only the links from the root of the subtree to each child that is to the right of child i (child $i + 1$ to child k) are severed.

parent 2 it might be in a location that is selected for inclusion in the parameter list, or selected by a selector operator. It is, therefore, now an exon. Similarly an exon in one parent may become an intron in another parent. Of course the exchanged subtrees may both be exons, in which case the crossover behaves identically as with chromosomes without introns; or the exchange may be between two intron subtrees, which will have no external effect.

Just as mutations may be suppressed inside an exon, so too may crossovers that take place exclusively among intron subtrees be suppressed. A crossover where only introns are exchanged, unlike mutation, actually can have a noticeable effect on the system. When only introns are crossed, the effect on the system is as if crossover did not occur, and the two parents are simply copied. This is identical to the problem of incestuous crossover, and similarly promotes premature convergence.

To prevent such crossovers, a method of detection is implemented. The simple labeling of genes as introns or exons is insufficient, for 1-point crossover, because an intron may lie inside the chromosome, with exons farther down the genome. These exons would be crossed if the intron's locus was selected as a crossover point (see Figure 6.3). Only if all introns to the left of (and including) the crossover point are introns is the crossover ineffective. These introns are called tail introns, since they are on the tail end of the chromosome. Any crossover that uses crossover points which are located at tail introns in both parents, will exchange segments that will not be expressed in either child, since they will remain introns (see Figure 6.4).

To label genes as tail introns, first label all genes as either introns or exons during decoding (the same method as used when preventing intron mutation), then, scan the chromosome from the end backwards (also see Figure 6.4). All introns encountered before the first exon is reached, are labeled as tail introns. Then, when a crossover point is picked, if the loci of both parents have a gene that is labeled as a trailing intron, a new

Figure 6.3 Crossover, When Both Crossover Points are Introns, Can Produce New Children

Legend:

Operator Set	F = {+, -, *, /, f, g, h}	Bold Symbols	Exons
Terminal Set	T = {1-9, i, j, k}	Light Symbols	Introns
	E - Exon I - Intron T - Tail Intron		

**Figure 6.4 Crossover When Both Crossover Points are Tail Introns:
No Effect on Phenotype**

crossover point is chosen.

The effect of uniform crossover on introns is similar to the one point crossover, except the distribution is more like a series of point mutations. Therefore the problem of premature convergence, because only introns have been crossed resulting in identical phenotypes, is negligible. So the analysis as done for mutation applies to uniform crossover, and no precautions have to be taken to prevent intron/intron crossovers.

6.3 INVERSION AND GENE JUMPING

6.3.1 INVERSION

The inversion operator was introduced early in the genesis of genetic algorithms. It was developed to place linked genes together to prevent disruption by the 1-point (or n-point) crossover. The inversion operator reverses a segment of the chromosome, but keeps the original order for decoding through the use of a template which records the original position of each gene ([Goldberg 89], pp. 166-167). When decoded, the

Figure 6.5 An Example of Inversion, as used in a Genetic Algorithm

chromosome is read in the order provided by the template. The template keeps track of the original order by mimicking the inversion process that was invoked on the chromosome. For an example of inversion (on a chromosome encoded with a binary alphabet) see Figure 6.6.

It was hoped that inversion would dynamically find the proper sequence that kept linked genes together, which would form optimal building blocks without having to know, a-priori, what the proper sequence for encoding the chromosome was. While inversion was an interesting idea, there have been no reports of it producing any significant improvements to any system that it has been applied to.

The inversion operator was inspired by a genetic operator in biology of the same name. The biological 'inversion' also inverts a segment of the genetic code, and yet keeps the original order of the genetic code during transcription³. The correct sequence is maintained, not through the use of a template, but because there is a special direction for transcription. This direction (caused by the asymmetry of the molecules that form the RNA and DNA strands) is reversed by the act of inverting the sequence. Therefore as long as the codon being transcribed is within the inverted segment, it will be translated into a correct protein.

The effect of inversion on the biological chromosome is, however, somewhat different from the effect produced by inversion used in a GA. Instead of trying to bring two linked genes close together, it is a mechanism that prevents crossover from occurring in the affected section. For this reason inversion is sometimes called a 'crossover inhibitor'. This is done through semi-infertility. During the pairing of

³ 'Transcription' is the process that reads the DNA, into the mRNA.
 'Translation' occurs at the ribosome, and turns the code from the mRNA into a protein.
 Both processes together are called 'Protein Synthesis'.

homologous chromosomes⁴ in the crossover step, which is called 'synapsis' in genetics, areas of similarity have great attraction. This causes a loop to be formed, where the inverted segment lies, in one of the chromosomes, with the other chromosome binding to the outside of the loop (this puts the code of the inverted segment into the 'correct order' to be matched with the other chromosomes code, see Figure 6.6 for an example).

However, if a crossover occurs inside the inverted segment, the ordering of the genes in the offspring is neither inverted, nor non-inverted, but mixed. This produces non-viable children (again, see Figure 6.6).

Figure 6.6 Inversion as a Crossover Suppressor: Crossover is shown occurring inside an inverted segment producing invalid offspring. Two examples are presented, with different effects (both lethal), based on the position of the centromere (a non-DNA feature in the Chromosome that bind the two DNA strands together). This figure is reprinted from [Stansfield 91] pg. 183.

⁴ Homologous chromosomes are chromosomes, from different parents, that have the same code, except for alleles.

Thus inversion 'prevents' crossover disruption within the inverted segment, through the attrition of genomes that were produced by such a crossover. A more exact analogy of inversion, for application to a GA, would use the GA 'inversion' template to label a segment of the chromosome as being 'protected' from crossover. No crossover point would be allowed to be selected from it.

While the effect of the inversion operator is different in GA from that used in Genetics, the goals behind them are actually very similar. They both are an attempt to prevent crossover from destroying important schemata (genetic sequences).

6.3.2 A MISLABELED GENETIC OPERATOR

There is another method that appears in some of the GA systems found in the literature, which is also called inversion, and is even thought of as being the same operator, just in a different setting [Fujiki & Dickinson 87],[Koza 92]. This 'inversion' operator is usually found applied to systems that have, encoded in the chromosome, segments of genetic code which together form an identifiable phenotypic trait when decoded, yet have some degree of positional independence inside the chromosome. In other words, the operator can be applied if the chromosome is built up of blocks of code that retain their meaning when moved to another area of the chromosome. This genetic operator appears superficially to be identical to inversion, since it also inverts a sequential order, but instead of inverting a sequence of single loci, they invert the sequence of the 'meaningful' building blocks. These 'meaningful' building blocks will be called 'transposons' for reasons to be discussed in the next section. A template is not used during this 'inversion'. It is not necessary, since each transposon retains its 'meaning' in any position. However, the general phenotype may change, since the transposon will

have a slightly different effect in the new position. For example, a sorting routine, if moved to a different area of a program, will change the outcome of the program (have a different phenotypic effect) yet still can be said to 'sort its input'. The change of the phenotype after the operator has been applied is the first indication that this 'inversion' operator is, in fact, not inversion. After an application of normal inversion, the phenotype remains unchanged.

This type of 'inversion' frequently appears in GA systems which are designed to solve program induction. The system created by Fujiki and Dickenson to produce programs to solve the 'prisoners dilemma' problem, uses inversion to reverse the order of conditions in the 'COND' statements, which were used as chromosomes [Fujiki & Dickinson 87]. The 'inversion' operator was generalized by Koza for GP and re-titled as the 'Permutation Operator', which not only would invert the ordering of the subtrees below a randomly chosen node, but also rearrange them in any permuted order [Koza 92] (this extension can also be applied to regular inversion as well, and is called a 'reordering operator' by Goldberg [Goldberg 89]).

Despite its frequent appearance in the literature, this inversion 'look-alike' operator has been routinely regarded as an application specific form of inversion. When Koza presented the Permutation Operator, he placed it in the context of inversion, and, then, dismissed it, since inversion does not seem to be useful to a GA in general⁵. However, this operator is not being used to prevent schema disruption, as is the intent of the regular inversion. Nor is this the effect that it has, in the applications that use it.

The 'inversion' operator has two potential effects. The first effect, which is usually the design consideration behind its use, is to try out a different ordering of the transposons, hoping that a different arrangement will have beneficial phenotypic effects.

⁵ It is probably true the Permutation Operator is not useful for Genetic Programming. Reasons will be given for this later.

The second effect is less obvious, but potentially the more potent of the two. The operator affects the distribution of transposons to other positions in the chromosome, where it might also be useful. The new placement in the current chromosome may not be very beneficial, it may even be detrimental, but from that position the transposon will be transferred to other chromosomes by crossover, alleviating the need to develop the code in those positions independently. To be effective in this way, it is imperative that the crossover will not disrupt the transposons.

It is difficult to view the first effect as having a large impact in the evolutionary process. There is no a-priori reason for the system to believe that a new ordering would, statistically, have a better effect on the fitness than by inserting new genes through crossover. With crossover in place, it is just a redundant operator in the system. There is no detriment in using it, but no benefit either. The second effect, however, is far more interesting. The schema being moved has a high probability of being useful, since it is from a highly fit chromosome (otherwise the chromosome it exists in would have probably not been chosen). This method alleviates the need for the same schema segment having to be re-evolved in each location where it is needed. This may be of great benefit for speeding up convergence in a GA.

6.3.3 TRANSPOSITION, A.K.A. JUMPING GENES

Since the 'inversion' operator is different from the normal inversion operator, in both implementation (no template used), and purpose, the technique should have its own name. Using the same term for both operations has already, in the past, created confusion, and allowed this operator to be either overlooked, or dismissed out of hand. A good term of reference should be one that is not currently being used in the GA literature,

and is descriptive of both the immediate behavior of the operator, and its emergent properties. One such possible term, which is again borrowed from genetics, is 'transposition', also known as 'gene jumping'.

The transposition is the movement of genetic material segments, called transposable elements, from one area in the chromosome to another. The code responsible for the transposition is usually found in the transposable element itself. The transposable element can just consist of the code necessary to attain its transposition to a different location in the chromosome, in which case it is called an **insertion sequence (IS)**; or it might contain other code not relevant to the transposition process, in which case it is called a **transposon**.

Transposition elements have been found in many different types of organisms. They were first discovered as the genes responsible for the differing colouration of corn, and has been well studied in *e-coli* bacteria. Therefore it exists in the chromosomes of both eukaryote, and prokaryote cells. In *D. melanogaster* they constitute 5-10% of the genome ([Hartl, et. al. 88], pg. 148). Transposition appears to be useful in bacteria populations where it may be responsible for the propagation of multiple-drug resistance ([Stansfield 91], pg. 321). This is accomplished, with the aid of natural selection, by rapidly spreading the good gene throughout the population.

There are two types of transposable elements. A transposable element is said to have a '**conservative mode**' if it moves to the new location without leaving a copy of itself behind. '**Replicative mode**' describes transposable elements that, after the jump, are in both the old and new positions. Usually transposons do not contain material normally essential for the survival of the organism. However they are very useful to organisms in harsh environments where mandatory changes can be spread swiftly throughout the population.

Biological transposition has many features in common with the GA operator in question, which makes it a useful analogy. They are both methods for spreading potentially beneficial segments of genetic code throughout the population; they both accomplish this by moving those segments to different parts of the chromosome; and the move may not be directly beneficial to the individual itself.

The analogy is, of course, not exact. The main use of transposition, as seen in genetics, seems to be the ability of transposons to easily combine traits developed in separate individuals, rather than to transfer information that has been found useful in one part of the chromosome to a different position where it had not yet developed. However the use of the term is only for analogical purposes; it is not a model of the process. The two processes do not have to be identical for an identification, which relates the two, to be practical. The inversion analogy, for example, is far from exact, but is still used.

The transposition analogy may also be helpful for suggesting modifications and alternatives to the GA operator: so far only an analog of a conservative mode transposition has been developed; perhaps an operator that works in replicative mode, by copying the transposon instead of just moving it, could be advantageous.

Since the operation is currently being referred to by a name that is being applied to a different process, and since it shares many traits with biological transposition, yet differs only slightly, naming the operator the 'transposition operator', and the process either 'transposition', or 'gene jumping', makes sense.

6.3.4 THE TRANSPOSITION OPERATOR: AN EXTENSION TO EP-I

The transposition operator can be applied to the EP-I system. However, it is only applicable to systems that use a depth first encoded chromosome, and an n-point

crossover operator. This is a result of the nature of transposition, which moves meaningful contiguous segments to other locations in the chromosome, for eventual distribution to other chromosomes through a non-disruptive crossover operator. The only encoding method that contains meaningful contiguous segments is the depth first encoding method; and the only crossover operators that keeps those segments together, given a depth first encoding, are the n-point crossover operators.

There are three general variations of the transposition operator that can be used. The first is called 'transposition between siblings'. A chromosome is selected using the same selection method used for all genetic operators, and a point is randomly chosen. This is not the root of a subtree to be transposed, but the parent of the various subtrees that will be transposed, hence the name 'between siblings' (see Figure 6.7). There are many variations for effecting the transposition. A subtree could be chosen, removed, then placed between two siblings (other than the two between which it was originally). Alternatively, two of the subtrees below the chosen node could be randomly selected, and the subtrees exchanged. A third variation, which is identical to the permutation operator in GP, randomly permutes all the children of the originally selected node.

The second transposition method, called 'invisible transposition between siblings', is the same as the first method (with all its variants), except that pointers are kept which define the order of evaluation of the children, and are exchanged as well as the subtrees (see Figure 6.7). The manipulation of the parameter pointers, is an identical technique as the manipulation of the order templates in the original inversion operator. This keeps the decoding of the parse tree unaffected, which in turn leads to no changes to the phenotype. Hence the use of the term 'invisible'. To implement this method, the chromosome encoding scheme with the Parameter Location Vector must be used. By maintaining the same phenotype, this variant of transposition has even stronger similarities, than the other methods, to inversion. However, it is still a form of

Figure 6.7 Invisible and Regular Transposition Between Siblings

Figure 6.8 Transposition Within Level

transposition, and not inversion, since the effect it has is to distribute code to different locations in the chromosome, and not to prevent the disruption of schemata.

The third transposition method is called 'transposition within level'. This is the same as transposition between siblings, except that the subtrees are allowed to be exchanged between any two nodes on the same level of the expanded parse tree (see Figure 7.8). All of the variations used in transposition between sibling are applicable to transposition within level. An 'invisible' transposition within level is not possible, since there are no pointers that can keep track of all nodes on a level, be permutable, yet still keep subtrees together in a chromosome. The children of any node, if such a scheme were put into place, could be under any other node of the same level, thus no contiguous subtrees can be formed.

Experimentation has to be done to see whether any of the various transposition methods are useful in EP-I. However, some hypotheses on the effects of the three methods, and their advantages, and disadvantages, in comparison to each other, follow. Transposition 'within level', distributes the transposons most widely, so if transposition proves to be useful, this is probably the best method. Between the two 'within sibling' methods, the 'invisible' method is probably superior because it guarantees that the offspring will have a good fitness (since the phenotype remains the same, the offspring must have a good fitness or the parent would not have been selected), and so be more likely to be selected for crossover, on average, than the 'visible' method, which changes its phenotype, and so can produce offspring with poor fitness. However, the 'invisible' method needs the Parameter Location Vector from the third encoding method, which requires more memory than the other encoding methods. Since memory is a tight constraint for EP-I, the increased convergence by the use of an 'invisible' transposition between siblings may not be worth the restriction of the size of the population, or tree height, that a reduced memory constraint would foster.

6.4 THE EP-I SIMULATION OF GP

6.4.1 THE GP-CROSSOVER

The EP-I algorithm can explicitly model Koza's GP algorithm. Underlying both the GP and EP-I algorithms is the genetic algorithm, and as such they can be uniquely defined by the phenotypic expression of the genes, the selector methods to choose individuals for reproduction, and the effects of the operators on the phenotype of the chromosome. In other words we can consider EP-I as a good model of GP if: the

chromosomes have the same phenotypic expression as in GP; the genes are manipulated in the same fashion by the genetic operators (decoding of the gene leads to the same phenotypic expression after the genetic operators are applied); and the selection methods for reproduction are the same. For GP the phenotype is produced by the evaluation of the parse tree, the primary genetic operator is GP's specialized crossover, and the selector methods are identical to those used in GA. EP-I uses chromosomes that decode to the same parse trees as used in GP (by design, see section 5.2.2), and uses the same selector methods. This leaves the crossover that needs to be emulated.

To emulate the GP style crossover, a genetic operator called 'GP-Cross' is introduced to EP-I. Like the regular GP crossover, the crossover points are chosen independently for the two parents, but a restriction must be placed: only nodes that are exons can be selected. The subtrees below the points are then traversed, following the selector chains, and copies of the non selector exons are placed into two temporary tree structures, called 'transposition vectors' or 't-vectors'. Once the subtrees have been traversed, the two, newly produced, t-vectors are identical to subtrees, from the corresponding GP parse tree, which have the corresponding values of the crossover points as the roots. If a crossover point is chosen inside a selector chain, in the expanded parse tree, then the t-vector would be identical to the subtree of the parse tree, with a root that has a corresponding value which is located at the end of the selector chain. Thus it ensures that crossovers occurs only between nodes that are directly represented in the original parse tree (see Figure 6.9, which is placed at the end of the chapter).

T-vectors are used because a direct exchange of subtrees is not possible. The two subtrees that are being crossed probably have different heights, and so would produce a child with an irregular structure. This destroys the fixed structure property of EP-I. If we directly exchange the subtrees without t-vectors, one of the subtrees would have terminals at a level less than the full height of the expanded tree, while the other would have

terminals at a level greater than the height of the expanded tree. The small subtree could be grown to fit in the expanded tree, but the larger subtree would have to be pruned to fit. Growing a parse tree is easy; just add selector chains, and randomly fill in the rest. Pruning is more complicated. Selector chains must be found and shortened, and extraneous introns must be removed. This is why the t-vectors are formed; it allows the subtrees to be exchanged in the most compact form, and therefore they just need to be grown.

If the subtree in a t-vector is too large to fit in the receptor slot of the other parent, then that child is rejected. This models the case, during a GP crossover, when a child grows larger than the maximum tree height, and is rejected.

Once the t-vectors are in place within the copied children, the selector chains are re-grown. To re-grow the selector chains, random selectors should be generated. The resulting offspring will have, as embedded parse trees, the same parse trees that would have been formed if the regular GP crossover were applied to the parent parse trees (again, see Figure 6.9).

If desired, the extra introns that have to be generated during re-growth, can be equal to the nodes of the original parent tree. This can be done, with no cost, by just copying the entire parent, placing the t-vector in the chromosome at the crossover point, and, then, growing the selector chains by overwriting the loci that are to be converted. There are no real advantages, however, to doing this since the GP crossover never exposes introns (i.e. it never turns an intron into an exon), and mutation is not used in GP. Therefore the introns that are there will never be expressed (unlike when regular crossover operators are used, such as 1-point crossover), so it does not matter what the values of the introns are.

6.4.2 A TIME COMPLEXITY ANALYSIS OF THE GP SIMULATION

To be considered a proper simulation, the GP-Cross should have the same worst case time complexity as the crossover in GP. The GP crossover takes $O(n)$ time, where n is the number of nodes in the largest parse tree that will be crossed. This result comes from the need to copy both trees for the offspring before the subtrees can be exchanged. The selection of the two crossover points take $O(\log n)$ since the nodes can be found using tree traversal, and the exchange itself is an $O(1)$ process; the pointers to the selected subtrees are switched. Therefore, overall GP crossover takes $O(n) + O(\log n) + O(1)$ which equals $O(n)$.

The GP-Cross used by EP-I to simulate the GP crossover, can be divided into 4 steps: reproduction of the parent chromosomes; production of the t-vectors; insertion of the t-vectors into the offspring; and the growth of the selector chains to place the terminals at the leaves of the expanded parse tree.

The reproduction of the chromosomes is the same as in GP crossover, $O(n)$. Although the number of nodes in the expanded parse tree is always equal to the greatest parse tree possible in GP, this does not affect the worst case complexity analysis, which uses the largest parse tree that can be produced by GP anyway. The production of the t-vectors is also $O(n)$. They are found by decoding the chromosome, which at most has to look at all the nodes only once, and is therefore, $O(n)$.

Inserting the t-vectors into the offspring has, once again, a time complexity of $O(n)$. There are, at most, n nodes in the t-vector. The t-vector consists of a single subtree, which is written contiguously in the depth first encoded EP-I chromosome. Therefore it can be written directly into the chromosome, locus by locus, which obviously has a time complexity of $O(n)$.

The last step, creating a selector chain to move the terminals to the leaves of the expanded parse tree, is an $O(\log n)$ procedure. Each link in the selector chain that is added, moves the terminal down one level. Since there are $O(\log n)$ levels, the step takes $O(\log n)$ time to complete. In total, the GP-Cross operator has a worst case time complexity of $O(n) + O(n) + O(n) + O(\log n)$, which equals $O(n)$, and so is the same as the original GP crossover.

It is true that the GP crossover has a lower constant in front of the $O(n)$ than EP-I. Furthermore, GP has a much lower average time complexity, since the average GP parse tree being copied is far smaller than the largest possible parse tree, which is what EP-I always uses. Fortunately, the purpose of a simulation is to duplicate the behavior of a system, not the speed. In this respect the GP-Cross behaves admirably. However if a simulation takes forever to complete, it cannot really be said to be simulating the target system. This is why the worst case time complexity is needed. It assures that the simulation can be completed in roughly the same time as the system being simulated. GP-Cross both duplicates the behavior of the GP crossover, and has the same worst case time complexity; it can therefore be considered a valid simulation.

6.4.3 SIMULATING GP

By using the GP-Cross, the EP-I can completely simulate the GP system. Any parse tree from GP can be embedded in an expanded parse tree. Therefore the initial population used in GP can be recreated in EP-I. From then on, both systems use the same evaluation and selection routines; and, with the GP-Cross, produce the same offspring during reproduction. So, for each generation, the population of the two systems (assuming that the same random number are generated for both) are identical. Therefore

EP-I can properly simulate GP. This property will be used to analyze the nature of the GP crossover in chapter 8 (section 8.2.4).

Figure 6.9 An Example of GP crossover simulation in EP-I

Figure 6.9 a) GP Parse Trees With Crossover Points Indicated

Parse Tree 1:

Parse Tree 2:

Results After GP Crossover:

Child1:

Child2:

Note: all trees are composed from terminal set $T = \{+, -, *, /, f, g, h\}$
and terminal set $T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, i, j, k\}$.

Figure 6.9 c) The Transposition vectors To Be Inserted Into The Reproduced Children

Figure 6.x d) The T-Vector From Parent 2 Inserted Into The Child From Parent 1 At The Crossover Point:

Figure 6.9 e) The Children Of The Expanded Parse Trees After The Crossover

Parse Tree 1

Expanded parse tree 2

- Notes:
- The exons are written with large bold characters, the introns are written with regular characters
 - Child 2 was produced in an analogous manner to the production of Child 1
 - Compare the exons in the above trees to the expected Children from the GP crossover in Figure 6.9 a; the original expanded parse trees, after the simulated GP crossover, have produced children that correctly encodes the corresponding values of the crossover children from the original GP system

CHAPTER VII: EXPERIMENTATION

7.1 INTRODUCTION

The EP-I system has been implemented on a specially created GA, written in COMMON LISP, and using the COMMON LISP Object System (CLOS). The new GA was needed because the most common GA systems available can only encode chromosomes with the binary alphabet, and EP-I requires symbolic alphabets for its encoding. COMMON LISP was chosen because of the explicit use, in the language, of parse trees, the powerful symbol manipulation capabilities of the language, the ability to evaluate programs created "on the fly", and for the object oriented paradigm that the CLOS facility provides. The system currently runs using the Macintosh COMMON LISP 2.0 interpreter (MCL) , on both the Macintosh II, and the Macintosh Quadra, under System 7. However since MCL is completely compatible with the current COMMON LISP standard, "COMMON LISP The Language, 2nd Edition" [Steele 90], the code is portable to most other COMMON LISP interpreters on many other platforms.

Two basic types of experiments are performed to test the EP-I system: tests on the feasibility of the system; and tests on its properties. The feasibility tests are applied to determine if EP-I can solve program induction problems, and if so, how it compares with GP. The rest of the experiments test some properties of both the system, and program creation through evolution, in general.

This chapter will first present various design elements that are used by the experiments: a summary of the test suite used; and the design options chosen when running the GA, such as the selection and reproduction methods used. Then the experiments are described, presenting both the purpose of the experiments, and the

method used in the experiments. Finally, the results of the experiments are presented: first a typical run is described; then the summary of the convergence rates, which evaluate the performance of the experiments, is listed. The implications of the results are discussed throughout chapter 8.

7.2 DESIGN ELEMENTS

7.2.1 SUMMARY OF THE TEST SUITES

The three applications used for the test suite are: symbolic regression; programming a 6 bit multiplexor; and inducing the recursive formula for the Fibonacci series. They have all been described in section 4.6. A summary of the system design used in EP-I to implement the applications is presented below. The summary includes: the goal of the application; the function set F , and the number of parameters each function takes; the terminal set T ; the test cases used; the standardized fitness used; and the EP-I specific design considerations, if any. The decoded chromosomes of EP-I are denoted as 's-expr($X = x_i, Y = y_j, \dots$)', where 'X', 'Y', etc., are the arguments taken by the s-expression, which correspond to the global variables used in the parse tree, and ' x_i ', ' y_j ', etc., are the initialization values for the global variables.

Symbolic Regression

Goal: Induce the formula $f(x) = x^4 + x^3 + x^2 + x$.

Function Set: $F = \{+, -, *, /, \sin, \cos, \exp, \log\}$; all the functions are protected functions as described in section 4.6; they take $\infty, 2, 2, 2, 1, 1, 1, 1$ arguments respectively.

Terminal Set: $T = \{X\}$.

Test Cases: 20 input values for X , x_1 to x_{20} , each randomly chosen between $[0, 1]$.

Standardized Fitness:

$$\sum_{i=1}^{20} \text{errors}_i, \text{ where } \text{errors}_i = |f(x_i) - s\text{-expr}(X = x_i)|.$$

Implementation Notes:

Since the terminal set consists of only one element, the genes that encode terminals will never change during the execution of EP-I. Therefore the terminals are not included as genes in the chromosome. They are re-inserted into their proper places, in the s-expression, during decoding. The tree height reported by the system is the height of the full expanded parse tree, but the number of loci in the gene is calculated without the terminal level. The length of the chromosome is, therefore, equal to chromosome length used by a regular application with the tree height reduced by 1.

Programming a 6 Bit Multiplexor

Goal: Induce the program:

```
(if a1
  (if a0 d3 d2)
  (if a0 d1 d0)),
```

where a_0 and a_1 are the address lines, and d_0 , d_1 , d_2 and d_3 are the data lines.

Function Set: $F = \{\text{and, or, not, if}\};$
they take 2, 2, 1, 3 arguments respectively.

Terminal Set: $T = \{a_0, a_1, d_0, d_1, d_2, d_3\}$.

Test Cases: All of the possible truth value combinations of a_0 a_1 d_0 d_1 d_2 d_3 . There are 64 such test cases.

Standardized Fitness:

The number of times the s-expression being evaluated produces the wrong result when applied to the various test cases. Since there are 64 test cases, the worst fitness score is 64 errors, and the best score is 0 errors. The average fitness of a program that produced random results would be 32.

Fibonacci Sequence Induction

Goal: Induce the recursive series definition

$$s_n = s_{n-1} + s_{n-2}, s_1 = 1, s_0 = 1.$$

Function Set: $F = \{+, -, *, \text{SRF}\}$; all the functions are protected functions as described in section 4.6; the SRF function is the self reference function also described in section 4.6; each function takes 2 arguments .

Terminal Set: $T = \{J, 0, 1, 2, 3\}$, where J is the sequencing index.

Test Cases: The values of the Fibonacci series, for $J = 0$ to 20.

Standardized Fitness:

$$\sum_{i=0}^{19} \text{errors}_i, \text{ where } \text{errors}_i = |s_i - s\text{-expr}(J = i)|$$

Implementation Notes:

In the implementation used for GP [Koza 92], the terminal J is inserted in the s-expressions, during initialization, in equal proportions as the sum total of the other terminals. This gives the system a greater number of the terminal J to use to form potential solutions. To simulate this in EP-I, there are 4 different symbols encoding J, while only one each for 0, 1, 2, and 3. This gives the same proportion for the appearance of the gene J, when compared to the other

terminals in the initial trees created, as in GP. This ratio also holds when considering which allele to use during mutation.

7.2.2 SYSTEM DESIGN OPTIONS

To improve the performance of the basic genetic algorithm, the Handbook Of Genetic Algorithms [Davis 91] recommends the use of linear rank selection (which is called linear normalization), and unique elitism (which is called 'steady state without duplicates'). EP-I is run with these design options. The rank selection parameter 'max' (see [Baker 85]) is set to the value 2.0, which maximizes the convergence rate.

Pure reproduction, which uses the selection method to select chromosomes to copy to the next generation, is not usually used in the experiments. Instead all direct copying of chromosome into the next generation is done during the elitism operation. All other positions in the new population are filled by the offspring produced by the crossover operator (or other genetic operators). This strategy is also one recommended in the Handbook Of Genetic Algorithms, and is, intuitively, a good strategy to use. To converge rapidly, you want good breeding stock. However, some of the genes in a bad individual might be useful, or even necessary, for convergence. Therefore, it is advantageous to utilize the genes from a poorly fit individual. This is why, in the original GA, an occasionally unfit individual is chosen for placement in the next generation. However, why should the entire chromosome of an unfit individual be promoted, when it is only part of it that may be useful? It therefore makes more sense to keep some of the genes through crossover, but not the entire chromosome through pure reproduction.

All of the above design choices are used by the system in all experiments except for Experiment 2 (see below).

7.3 THE EXPERIMENTS

There are four general experiments which were run. Experiments 1 and 2 study the feasibility of the EP-I system. Experiment 1 tests whether program induction is possible using EP-I, and Experiment 2 compares EP-I with GP. Experiments 3 and 4 each tests a different property of the EP-I system. Experiment 3 changes the crossover method, from 1-point crossover to uniform crossover, to observe the effects of increasing disruption on the various subtrees in the parse tree (which may form the integral units for program manipulation). Experiment 4 tests the hypothesis that EP-I, unlike GP, will always produce a solution to the target problem, if run for as many generations as needed.

The purpose and methods of the experiments are given below. A successful run for any experiment occurs when a program is found with a standard fitness of 0 (except for experiments that perform symbolic regression which considers any fitness below 10^{-5} as a success due to round-off errors). The parameter settings for the experiments are summarized in Tables 7.1.

Experiment 1

Purpose: Determine if EP-I can solve program induction as a normal GA.

Method: Run the pure EP-I (with no extensions such as the transposition operator or the GP-Cross) on the test suite.

Symbolic regression and the programming of the 6 bit multiplexor problems are each run a few times (symbolic regression is run twice, the

multiplexor is run 3 times) with various crossover probabilities. This was done to experiment on the effect that different rates of crossover have on the probability of convergence. An extensive test of this parameter was not done because of time limitations.

Experiment 2

Purpose: Compare the convergence ability of EP-I with those previously found for GP.

Method: Use the results of experiment 1, and compare with the published results for GP when run on the same problems. Since all published results for GP on the problems in the test suite only use fitness proportional selection with no elitism, choose one of the problems in the test suite, and re-run it under those conditions for direct comparison.

Symbolic regression was chosen as the sample problem. Two different settings for the crossover probability were used to compare with similar tests performed in experiment 1.

Experiment 3

Purpose: To study the effects of disrupting the subtrees of the parse tree, which are the suspected natural building blocks for GA systems that solve program induction using parse tree encoding methods .

Method: Use uniform crossover on one of the problems in the test suite. Uniform crossover provides a mechanism to change the probability of disrupting high order schemata. The probability of disrupting high order schemata is in direct proportion to the value of the gene exchange probability used. Using this mechanism, different sized subtrees are allowed to survive or

be disrupted. Vary this parameter to see if larger values destroy the ability of the system to converge.

The 6 bit multiplexor was used for the sample problem. The parameter that controls the probability of loci exchange was set to $P_0 = 0.2$ as a base value. This is the setting that was most successful in the experiments performed by Spears and De Jong [Spears & De Jong 91]. It was then increased to 0.3 and 0.4 (a setting of 0.5 was also prepared, but was not run for lack of time).

Experiment 4

Purpose: To determine whether EP-I can guarantee convergence if it is allowed to run for an unlimited number of generations.

Method: Run the EP-I on one of the problems in the test suite for an unlimited number of generations.

Symbolic regression was chosen for the test problem. A population size of 200 was used, instead of 500, to decrease the time taken per generation. This slows convergence, but since the run is allowed to continue for a very long time, this should not be a hindrance. It is possible that the reduced population could bring the system below a threshold where convergence is not possible. However some preliminary experiments showed almost as good convergence for a population size of 200, as for a population size of 500, when solving symbolic regression, so this is not a worry. To simulate a run with a nearly unlimited number of generations, the EP-I system was setup to perform thousands of generations per run. The system was run with five different settings of the genetic operator parameters.

Table 7.1 Parameter Settings: Experiments 1 and 2

		Mut. Prob.	Xover Prob.	Repr. Prob.	Elite %	Xover Type	Elite Mode	Tail Intron	Select ⁿ Mthd	Pop ⁿ Size	Gen. Limit	Tree Ht.
Experiment 1												
Symb. Regr.	a1	.06	.8	0	.2	1pt	U	Y	RS	500	51	4
	a2	.06	.9	0	.1	1pt	U	Y	RS	500	51	4
6 Bit Mplx.	b1	.05	.9	0	.1	1pt	U	Y	RS	500	51	4
	b2	.05	.8	0	.2	1pt	U	Y	RS	500	51	4
	b3	.05	.6	0	.4	1pt	U	Y	RS	500	51	4
Fibonacci	c1	.05	.8	0	.2	1pt	U	Y	RS	500	51	3
	c2	.05	.8	0	.2	1pt	U	Y	RS	500	51	4
Experiment 2												
Symb. Regr.	a1	.05	.7	.3	0	1pt	N/A	N	FPS	500	51	4
	a2	.05	.5	.5	0	1pt	N/A	N	FPS	500	51	4
Experiment 3												
6 Bit Mplx.	a1	.05	.9	0	.1	Unif .2	U	N/A	RS	500	51	4
	a2	.05	.9	0	.1	Unif .3	U	N/A	RS	500	51	4
	a3	.05	.9	0	.1	Unif .4	U	N/A	RS	500	51	4
Experiment 4												
Symb. Regr.	a1	.06	.75	0	.25	1pt	R	Y	FPS	200	no limit	4
	a2	.06	.6	0	.4	1pt	R	Y	FPS	200	no limit	4
	a3	.05	.5	.5	0	1pt	R	Y	FPS	200	no limit	4
	a4	.06	.7	0	.3	1pt	R	Y	FPS	200	no limit	4
	a5	.06	.5	0	.5	1pt	R	Y	FPS	200	no limit	4
sums to 1												

Legend:

RS - Rank Selection

1pt - One Point Crossover

U - Unique Elitism

Y - Yes N - No

FPS - Fitness Proportional Selection

Unif - Uniform Crossover

R - Regular Elitism

N/A - Not Applicable

7.4 RESULTS

7.4.1 A DESCRIPTION OF A SUCCESSFUL RUN

A typical successful run is presented in Table 7.2. It is the second run produced for Experiment 1-b1 (the 6 bit multiplexor programming problem), and the first correct solution. There is a maximum of 51 generations that may be evaluated each run. A solution was found on run 30, so the run was terminated and never reached the 51st generation. Each line in the run presents: the number of the current generation; the standardized fitness of the highest ranking member in the population; the average standardized fitness of the population; and the program produced by the highest ranking member.

For the 6 bit multiplexor problem, the standardized fitness ranges from 0, which is a perfect score (no wrong data line values produced), to 64, which means all data lines were set incorrectly. If a system would just guess randomly, one would expect a standardized fitness of 32 to be produced.

In the initial, randomly produced population, the average fitness is only slightly better than that predicted for a random result: 29.5, as opposed to 32. The fittest individual found by the system produced only 16 incorrect results. The program corresponding to that individual is '(IF A0 D3 D2)'. This very simple program always produces the right result if A1 is set to true, but will be wrong 50% of the time if A1 is false. This matches the fitness observed.

In the succeeding generations, the fitness begins to improve. At first only the average fitness rises, as poorer results are ignored and better results, obtained through both crossover and direct reproduction, are promoted. New solutions as good as the

original fittest member are produced, and are seen as the new fittest member (when there are ties, the fittest member is randomly selected). By generation 5 a better individual has been produced, and becomes the fittest member, to be quickly succeeded in generation 6. This general progress continues, with new plateaus reached, and then exceeded, a few generations farther.

During this run, the average fitness climbs in a monotonic fashion. Each generation is better, on average, than the previous one. This does not always happen; especially if the process stalls. In these cases, the average fitness will fluctuate around a sticking point, until a new solution, which is better than the current maximum, is found, then the system resumes climbing again.

By generation 22, a good general outline for the final program can be seen forming, with a correspondingly low error rate of 4. Finally, in generation 30, the correct solution is found:

```
(IF A0
  (IF A1 D3 D1)
  (IF A1 D2 D0)).
```

In this case the solution is both correct, and optimally coded. This is not always the case. In the very next run, a correct solution was again found, but this time it was not optimally encoded. The solution produced was:

```
(IF (NOT A0)
  (IF (AND A1 A1)
    D2
    D0)
  (IF (AND A0 A1)
    (IF A1 D3 D0)
    (IF A0 D1 D2))).
```

This is, of course, the correct solution, although it is probably easier to check its validity through a truth table, than through inspection.

Table 7.2 Average Run for 6 Bit Multiplexor Problem

Gen	Max. Std. Fitness	Average Std. Fitness	Fittest Program	Gen	Max. Std. Fitness	Average Std. Fitness	Fittest Program
0	16	29.519	(IF A0 D3 D2)	9	12	22.715	(AND (OR D2 A0) (IF A1 D3 D1))
1	16	27.407	(IF A0 D3 D2)	10	12	22.105	(AND (OR D2 A0) (IF A1 D3 D1))
2	16	26.468	(IF (NOT A1) (OR A0 D0) D3)	11	10	22.011	(IF (IF D3 (OR D2 D0) (AND D1 A0)) (IF A1 D3 D1) (IF A0 D3 D0))
3	16	26.045	(IF (NOT (NOT A0)) (IF D3 D3 D1) (IF (NOT A0) (OR D2 D0) D3))	12	8	21.465	(IF (AND (AND A1 A0) (IF D1 A1 A0)) (IF (NOT A1) (OR A0 D0) D3) (IF A1 D2 D0))
4	16	25.822	(IF A0 D3 D2)	13	8	21.021	(IF (AND (AND A1 A0) (IF D1 A1 A0)) (IF (NOT A1) (OR A0 D0) D3) (IF A1 D2 D0))
5	14	24.962	(AND (IF (OR D2 D0) (IF A1 A1 D1) (IF A0 D3 D0)) (OR D3 (NOT A1)))	14	8	20.068	(IF (AND (AND A1 A0) (IF D1 A1 A0)) (IF (NOT A1) (OR A0 D0) D3) (IF A1 D2 D0))
6	12	24.498	(IF (NOT A1) (IF D2 D1 D2) (IF A0 D3 D2))	15	8	19.665	(IF (AND (AND A1 A0) (IF D1 A1 A0)) (IF (NOT A1) (OR A0 D0) D3) (IF A1 D2 D0))
7	12	23.906	(IF (NOT A1) (IF D2 D1 D2) (IF A0 D3 D2))	16	8	18.573	(IF (AND (AND A1 A0) (IF D1 A1 A0)) (IF (NOT A1) (OR A0 D0) D3) (IF A1 D2 D0))
8	12	23.129	(IF (NOT A1) (IF D2 D1 D2) (IF A0 D3 D2))	17	8	17.714	(IF (AND (AND A1 A0) (IF D1 A1 A0)) (IF (NOT A1) (OR A0 D0) D3) (IF A1 D2 D0))

Continued on next page . . .

Table 7.2 continued Average Run for 6 Bit Multiplexor Problem

Gen	Max. Std. Fitness	Average Std. Fitness	Fittest Program	Gen	Max. Std. Fitness	Average Std. Fitness	Fittest Program
18	8	17.122	(IF (AND (AND A1 A0) (IF D1 A1 A0)) (IF (NOT A1) (OR A0 D0) D3) (IF A1 D2 D0)))	25	4	14.298	(IF A0 (IF A1 D3 D1) (OR (IF A0 D3 D0) (IF D2 A1 D2)))
19	8	16.326	(IF (AND (AND A1 A0) (IF D1 A1 A0)) (IF (NOT A1) (OR A0 D0) D3) (IF A1 D2 D0)))	26	4	14.289	(IF A0 (IF A1 D3 D1) (OR (IF A1 D3 D0) (IF D2 A1 D2)))
20	8	16.310	(IF A0 (AND (IF A1 D3 D1) A0) (IF A0 D3 D2))	27	4	13.246	(IF A0 (IF A1 D3 D1) (OR (IF A0 D3 D0) (IF D2 A1 D2)))
21	8	16.242	(IF A0 (AND (IF A1 D3 D1) A0) (IF A0 D3 D2))	28	4	13.133	(IF A0 (IF A1 D3 D1) (OR (IF A0 D3 D0) (IF D2 A1 D2)))
22	4	16.236	(IF A0 (IF A1 D3 D1) (OR (IF A0 D3 D0) (IF D2 A1 D2)))	29	4	12.381	(IF A0 (IF A1 D3 D1) (OR (IF A0 D3 D0) (IF D2 A1 D2)))
23	4	15.665	(IF A0 (IF A1 D3 D1) (OR (IF A0 D3 D0) (IF D2 A1 D2)))	30	0	11.482	(IF A0 (IF A1 D3 D1) (IF A1 D2 D0))
24	4	14.845	(IF A0 (IF A1 D3 D1) (OR (IF A0 D3 D0) (IF D2 A1 D2)))				

7.4.2 A SUMMARY OF THE PROBABILITY OF CONVERGENCE FOR THE EXPERIMENTS

No experiment was performed with only one run. Since there is a probability, sometimes even a high probability, that any individual run may fail, many runs have to be tried to guarantee success. The measure of the probability that the system will converge, called the **probability of convergence**, or the **convergence rate** (see section 4.5), is denoted in the table of results as P_c . It is this measurement that determines how easily the system can find a solution to the problem; hence it is the proper measurement to use to evaluate the experiments. The measured average number of successes is, of course, not the actual probability of convergence, but only an estimate of it. As with all estimates, it has a corresponding margin of error (again see section 4.5), which is also presented in the results.

The results of the experiments are summarized in table 7.3. The table lists the experiment number, number of successes, an estimation of the convergence probability P_c , and the error in the estimation. To find the convergence rate estimate, approximately 30 runs were used on each test, with two exceptions. Experiment 4 runs the system for an unlimited number of generations, to see if it will always converge; here only 1 run, per setting, was produced, since the probability of convergence is not applicable. The second exception is Experiment 1-c, finding the Fibonacci recurrence relation; over 100 runs were used because of the low probability of convergence for this problem.

Table 7.3 Summary of Results

		Runs	Successes	P_i	Error
Experiment 1					
Symb. Regr.	a1	30	14	.47	$\pm .18$
	a2	30	11	.36	$\pm .16$
6 Bit Mplx.	b1	30	22	.73	$\pm .16$
	b2	30	18	.6	$\pm .18$
	b3	30	16	.53	$\pm .18$
Fibonacci	c1	113	11	.1	$\pm .06$
	c2	200	16	.08	$\pm .04$
Experiment 2					
Symb. Regr.	a1	30	2	.07	$\pm .09$
	a2	30	8	.27	$\pm .16$
Experiment 3					
6 Bit Mplx.	a1	30	20	.67	$\pm .17$
	a2	33	12	.36	$\pm .16$
	a3	30	9	.3	$\pm .16$
Experiment 4					
Symb. Regr.	a1	1	0	N/A	N/A
	a2	1	0	N/A	N/A
	a3	1	0	N/A	N/A
	a4	2	1	N/A	N/A
	a5	1	0	N/A	N/A

CHAPTER XIII: DISCUSSIONS AND FUTURE WORK

8.1 THE PROPERTIES OF EP-I AND THEIR IMPLICATIONS

8.1.1 AN UNMODIFIED GA CAN PERFORM PROGRAM INDUCTION

The main goal of the thesis is to develop an implementation of a GA that could perform program induction. The first experiment applied the EP-I to a test suite of program induction problems which were originally used to test GP. The results came back positive. Solutions were found for all three problems. Symbolic Regression was solved 1 out of every 2 runs, and a correct 6 Bit Multiplexor program was generated 3 out of every 4 runs. For the production of a recursive definition of the Fibonacci sequence, the results look poor at first glance, however, Koza reports that GP finds a recursive definition only 22% of the time when using a population size of 1000. The problem was only run in EP-I with a population size of 500, thus it is not unreasonable that convergence should occur with less frequency.

Over all, EP-I obtained comparable results to GP on the same problems. On symbolic regression, and programming the 6 bit multiplexor, the raw average of success was actually greater than GP. For symbolic regression, EP-I has $P_s = .47$, versus $P_s = .35$ for GP. On programming the 6 bit multiplexor, EP-I obtained $P_s = .73$, while GP obtained $P_s = .68$. These results are deceptive, since the GP results are within the experimental error bounds of the probability of convergence found for EP-I. Furthermore, neither GP nor EP-I have been optimally tuned, for the various applications, so a direct comparison is impossible.

All of the problems solved, during experiment 1, were solved using rank selection and unique elitism. GP, however was run using 1-point crossover, without elitism. To determine whether the use of more powerful selection techniques were unduly affecting the results, experiment 2 was run. Symbolic regression was implemented and solved on the EP-I system using the same selection routines as the experiments done in GP: fitness proportionate selection without elitism. The result is much worse when compared to those obtained by the EP-I system using rank selection with elitism. The performance of EP-I with these settings is also worse than those seen in GP, however, the GP probability of convergence for symbolic regression is still within the error bound of the experiment. Also, as implied by the far lower result from experiment 2-a2, different settings can radically change the convergence rate of the system. The various parameters were not optimally chosen, when initializing experiment 2-a1, so having the GP probability of convergence within the error bounds of the experiment is satisfactory. Therefore GP and EP-I give similar results on the test suite.

Overall, the EP-I system, using the canonical GA approach, solves program induction with comparable performance to GP. This shows that there is nothing 'magical' about the GP crossover. Furthermore, the two systems achieve remarkably similar results, when run on the same problems, using the same function and terminal sets. Yet the crossover techniques used are vastly different, which one would think should lead to different dynamics for the two systems, and consequently different results (a third crossover, uniform crossover, also produced a similar probability of convergence when programming the 6 bit multiplexor in experiment 3-a). That the two systems behave so similarly when using the same primitives, leads to the possible conclusion that there is some underlying property to program design, which affects the ability of a GA system to evolve the programs. What these properties may be is unknown. Furthermore, the normal intuitions, when applied to the design of the various problems solved by GP, lead to

counter-intuitive notions about their convergence probability. Why is it, that a modular design, as found in solving the regression definition of the Fibonacci sequence, will converge only 20% of the time, while the block stacking problem (not yet performed using EP-I) almost always converges, yet it is designed with massive side effects, and huge order dependencies? Exploring this unexpected effect could prove to be an intriguing subject for future work.

8.1.2 EP-I IS NOT A RANDOM SEARCH

One of the typical arguments against the GA, is the accusation that the GA is only doing a random search. In other words, the results obtained by the GA is the product of blind luck. The proponents of the GA, however, claim that, although probability is involved in the algorithm, the results are produced by a directed search through the solution space, and not a random one. The same attack and defense is applicable to the solution of program induction problems in EP-I. If EP-I were indeed just a complicated random search, then an undisguised random search should do equally as well. However, EP-I starts out with just such a random production of solutions. There are 323 runs performed in experiments 1, 3, and 4. Each has a population size of 500. 60 additional runs with population size of 200 were run for experiment 2. This means that 173 500 potential solutions were created randomly in generation 0. Additionally 462 000 potential solutions were randomly generated in the many earlier runs that were performed to test the system (all of these were attempting to perform symbolic regression). Of these, not one found a solution. For symbolic regression, which has a probability of convergence $P_s = .47$, only 6 runs are needed to be 99% sure that a successful solution is found (the equation for predicting the number of runs needed to find a solution, was

presented in section 4.5). This means that no more than 153 000 individuals have to be processed to find a solution (6 runs, times 51 generations, times 500 members in the population). Yet, the first generation of the test runs generated 462,000 potential solution, this is over 3 times the number of individuals EP-I needs for a guaranteed solution, and no solution was found. From this it is easy to conclude that the success of EP-I is not a product of a random search.

8.1.3 SUNDRY OBSERVATIONS ON THE EXPERIMENTS AND ANECDOTES

Experiment 4 was designed to test whether the EP-I system would always converge if run long enough. This would have been a nice property to have, and one which GP does not enjoy. However, the experiment showed that this is not the case. In all the runs produced, each one failed to converge after running for thousands of generations.

When observing the results obtained from the different crossover probabilities used in the test suite, a mixed message was received. In Experiment 1-a (which solves symbolic regression), the 0.8 setting for the probability of crossover had a better convergence rate than the 0.9 setting. However in Experiment 1-b (programming the 6 bit multiplexor), the 0.9 setting was superior. In all cases, the values were within each others error bounds, so much larger experiments would be needed to settle the question.

There is also a difference found for the setting of the probability of crossover between the experiments that used rank selection with elitism (experiment 1), and those that used fitness proportional selection with no elitism (experiment 2). Experiment 2 results show better performance on runs with the crossover probability set lower (at 0.5), rather than higher (at 0.7). In experiment 1, the poorest results were obtained, for the 6

bit multiplexor, with a probability of mutation of 0.6. All other settings were used at or above 0.8. This implies that a higher crossover rate should be used when using Rank Selection with elitism, than when using Fitness Proportional Selection without elitism. However the analysis is only suggestive, not conclusive. The effect being seen might be a difference between the behavior of EP-I in solving the 6 bit multiplexor problem, versus solving symbolic regression, (with symbolic regression performing better at lower crossover rates), rather than a difference between rank and fitness proportional selection, or elitism and non-elitism. Furthermore, the error bounds overlap for most of the experiments (although there is a decisive difference between the performance of symbolic regression with a 0.8 crossover rate using rank selection and elitism, versus its performance with a 0.7 crossover rate with fitness proportional selection and no elitism). Again, more experimentation will need to be done, to resolve the question.

Finally, an anecdotal warning to other experimenters. It is tempting to run a very small number of runs to find the correct parameter settings, before proceeding with an experiment (such as five runs per parameter setting). This is because of the large amount of time it takes a GA, running a program induction application, to evaluate the population for each generation. However, because the typical probability of convergence is not in the high 90% range, large error bounds are produced. For example with 5 runs¹ on an experiment that has a probability of convergence of $P_c = .5$, the error bound is $\pm .62$, which covers the entire range of probabilities. This makes any result obtained totally meaningless.

¹ When less than 30 experiments are performed, the Z score cannot be used. The Student's T score is used in stead. Therefore, to calculate the error bounds with 95% certainty, $t_{0.25} = 2.78$ is used instead of $z_{0.25} = 1.96$, when only 5 experiments are run (degree of freedom of 4). See [Mendenhall, et. al. 81] for further details.

8.2 COMPARING GP WITH EP-I

8.2.1 GENERAL DIFFERENCES

Since EP-I uses a standard GA to perform program induction, it inherits all the properties of the GA, including the differences to GP discussed in section 4.4. A summary of the differences is presented in Table 8.1.

An apparent advantage that GP has over EP-I is its lower average memory requirement. The worst case space complexity of the two algorithms is identical; it occurs when the program, encoded in the parse tree, is expressed using the

Table 8.1 Differences between GA and GP

	GA	GP
Chromosome Structure	Linear	Tree
	Fixed	Variable
Loci Comparison Between Chromosomes	Comparable	Incomparable
Allele Definition	An alternate gene for the same locus	Not definable
Crossover	Exchanges genes at the same loci	Exchanges subtrees below 2 different nodes
Incestuous Crossover	offspring identical to parents	offspring are unique
	promotes premature convergence	prevents premature convergence
Mutation	Necessary for convergence	Unnecessary For convergence

full tree. The average space complexity, however, is much lower, for GP, than this upper bound. For EP-I, the average space complexity is equal to the worst case space complexity. This advantage is mitigated by the differing dynamics of the two systems. GP needs room for the parse trees in the population to grow and shrink. This is how it finds the solutions. In Koza's experiments, this parameter was set to a height of 17 ([Koza 92], pg. 114). EP-I, however, does not work by manipulating subtrees, but through manipulation of the alleles at each locus. The chromosome does not need to grow and shrink, to sample the various schemata in the system. Therefore, the height of the expanded parse tree can be set equal to, or only slightly larger than, the optimal encoding for the solution. For all the problems used in the test suite, a tree size of 4 seems to produce satisfactory results. Therefore EP-I may actually need less memory, in practice, than GP.

Another apparent advantage of GP over EP-I, is the consequence of incestuous crossover leading to the prevention, rather than promotion, of premature convergence. Fortunately, in EP-I, experiments show that premature convergence is not a problem when the population size is large enough. Table 8.2 presents the maximum and average adjusted fitness for Experiment 4-a4. This experiment was the one where runs were allowed to continue for an unlimited number of generations. Fitness Proportional Selection was used in the experiment, because it is much more prone to premature convergence than is Rank Selection. Since Fitness Proportional Selection bases its selections on the adjusted fitness values, instead of the standardized fitness values, the adjusted fitness values are given in the table. The experiment performs symbolic regression on a population of size 200, with the mutation probability set to 0.06, a crossover probability of 0.7 and the percentage of elite promoted to the next generation set at 30%. If premature convergence occurs, the average fitness would become equal to

Table 8.2 The Maximum and Average Adjusted Fitness for Selected Generations from Experiment 4-a4

Gen.	Maximum Adj. Fitness	Average Adj. Fitness	Gen.	Maximum Adj. Fitness	Average Adj. Fitness
0	0.2969	0.0558	15	0.3726	0.2524
1	0.2969	0.0748	16	0.3726	0.2519
2	0.2969	0.0838	17	0.3726	0.2673
3	0.2969	0.0997	18	0.3726	0.2817
4	0.2969	0.1052	19	0.3726	0.2681
5	0.3279	0.1212	20	0.3726	0.2574
6	0.3572	0.1298	21	0.3726	0.2601
7	0.3572	0.1410	22	0.3726	0.2634
8	0.3726	0.1648	23	0.3726	0.2633
9	0.3726	0.1844	24	0.3726	0.2638
10	0.3726	0.1783	25	0.3726	0.2805
11	0.3726	0.2043	26	0.3726	0.2740
12	0.3726	0.2220	27	0.3726	0.2574
13	0.3726	0.2303	28	0.3726	0.2783
14	0.3726	0.2393	29	0.3726	0.2771
⋮					
660	0.7267	0.5170	665	0.7267	0.5202
661	0.7267	0.5679	666	0.7267	0.5219
662	0.7267	0.5207	667	0.7267	0.5533
663	0.7267	0.5036	668	0.7267	0.4984
664	0.7267	0.5413	669	0.7267	0.5208
⋮					
3205	0.7866	0.5618	3212	0.7866	0.5167
3206	0.7866	0.5257	3213	0.7866	0.5243
3207	0.7866	0.4899	3214	0.7866	0.5429
3208	0.7866	0.5371	3215	0.7866	0.5216
3209	0.7866	0.5258	3216	0.7866	0.5514
3210	0.7866	0.5340	3217	0.7866	0.5184
3211	0.7866	0.5112	3218	0.7866	0.5563

the maximum fitness. At the start of Experiment 4-a4, the average fitness is far smaller than the maximum fitness, but is rising quickly. This stabilizes by generation 17, where it reached a fairly high average fitness in comparison to the maximum fitness, but approximately 10% to 15% lower. Since the adjusted fitness (which ranges from 0 to 1) is not linear in its growth, this difference is actually greater than it seems. The average after that seems to fluctuate higher or lower by about 2%, and seems stable. It certainly

does not seem to be approaching the maximum fitness, as it would if one individual had taken over the population. This result is seen again and again throughout the run. When a greater maximum fitness is found, the average starts to increase, but levels off at a value about 10% to 20% lower than the maximum. In no case did premature convergence occur.

8.2.2 THE SCHEMATA USED BY EP-I

In the EP-I system, both the GP style of schemata, and the Holland style of schemata can be applied. The Holland style of schemata is defined as a subset of chromosomes in the population that have the same genes, at specific loci. This definition is not usable for GP, since a node in the GP parse tree does not have a fixed location. This renders the comparison of loci meaningless. However, since EP-I chromosomes have a fixed structure, the Holland schema definition can be applied. The GP style of schemata can also be used in EP-I, especially in a depth-first encoding. The EP-I has, encoded in its chromosome, the expanded parse tree, which is a true parse tree. Therefore a set of subtrees can be defined across the EP-I population of parse trees in just the same manner as it is in GP (see section 4.7 for an explanation of the GP style schemata).

Based on the use of traditional crossovers in EP-I, the Holland style schemata definition is probably more useful in analyzing the behavior of the system, rather than applying the GP schemata definition. This is not only a more appropriate choice, it is also a very beneficial choice. The theory, derived for the regular GA using the Holland style schemata, is very extensive, while the theory developed for GP is virtually non-existent. Not much is known of the behavior of the GP schemata as applied to GP, which it was designed for, and how they are affected by the EP-I system is completely

unknown. Since EP-I allows both schemata definitions to be applied, it enjoys the best of both worlds. All of GA theory is inherited by the system, and now can be applied to program induction problems; and EP-I also allows the new GP schemata definition to be used on a system other than GP, perhaps allowing further insights into the behavior of this new schemata definition, and even possibly leading to a better understanding of GP as well.

8.2.3 ARE SUBTREES NATURAL BUILDING BLOCKS FOR PROGRAM INDUCTION?

The natural building block for GP, as identified by Koza, is the small subtree, which is a part of the set of subtrees that make up the GP schemata. The rationale is that GP crossover has a lower probability of disrupting these small subtrees. Therefore, the selection process will promote small subtrees which enhance the fitness of the chromosome that they are in, while winnowing out those that have deleterious effects. This is exactly the same as what happens to the short defining length building blocks in a regular GA system.

Keeping the principle of meaningful building blocks in mind (see section 2.3), an obvious question arises: are these subtrees vital to the encoding of a successful GA that solves program induction? Since GP was the only system that solved program induction problems over a wide range of problems, the question could not be answered, as there were no other systems for comparison. EP-I adds another 'point on the graph'. In fact it adds many, since many different crossovers and encoding methods are possible using the EP-I system.

When the EP-I system is used with a depth-first encoding of the chromosome, the subtrees of the GP system appear as contiguous loci. Although the selector operators can spread the subtree into a larger area, generally small subtrees are found in small contiguous portions of the chromosome. This means that the natural building blocks found for the 1-point crossover have as a subset the natural building blocks suggested by GP. This is only an approximate mapping since the building blocks favored by 1-point crossover can cross subtree boundaries (see Figure 8.1). However, those schemata that do cross the boundaries, may or may not be useful to the system for advancing convergence, and experiments would have to be designed to determine which of the possible building blocks are important.

The results from Experiment 3, while not fine enough to determine if the subtree is the vital building block in the system, suggest that it might be. This experiment examined the effects, on the EP-I system, of uniform crossover, where differing probabilities of gene exchange at a locus were tried.

Uniform crossover has the peculiar property of treating all schemata equally, regardless of their defining lengths. However, it does not treat schemata with different orders (the number of fixed positions in the similarity template) equally. Lower order schemata are less likely to be disrupted than higher order schemata. However, for schemata with the same order, the location in the chromosome of the fixed positions is irrelevant to the probability of disruption. Since small subtrees have low order, they are favoured for survival when the uniform crossover is performed. The probability of gene exchange raises or lowers the probability of higher order schemata surviving in an inverse manner. The higher the probability of gene exchange, the lower the probability of high order schemata surviving a crossover (see Figure 8.2).

The small subtrees, which are the building blocks of GP, are schemata with low order in the EP-I system, and so are not likely to be disrupted. However as the

Figure 8.1 The Building Blocks Favored by 1-Point Crossover:
Crossing Subtree Boundaries

	Number of Trees				
Tree Size	in Chromosome	Undisturbed	Undisturbed	Undisturbed	Undisturbed
3	8	5	4	2	0
7	4	1	1	1	0
15	2	0	0	0	0
		0.1	0.2	0.35	0.5
Probability of Gene Exchange					

Figure 8.2 The Higher The Probability of Gene Exchange in Uniform Crossover, The Higher The Probability That Subtrees Will Be Disrupted

probability of gene exchange increases, the size of the subtree that can survive, decreases. Therefore, if the subtree is an important building block, one would expect that as the probability of gene exchange is increased, the rate of convergence should drop. This is seen quite dramatically. When the probability of gene exchange was raised from 0.2 to 0.3, the rate of convergence went from $P_c = 0.67$, which is as good as the results obtained with 1-point crossover, down to $P_c = 0.36$, and dropped again to $P_c = 0.3$ when a 0.4 probability of gene exchange was used. This seems to support the notion that the subtree is an important building block for program evolution.

However, the above experiment is only suggestive, and does not conclusively determine how important the subtree is. Other factors that could be dependent on the order of the schemata may have been affected by the experiment. Further experimentation, using different crossovers, and different embedding methods for coding, are needed to answer the question.

8.2.4 ANALYZING GP THROUGH EP-I SIMULATION

Since EP-I produces similar results to those produced by GP, the simulation of the GP system, by adding 'GP Cross' to the EP-I system as described in section 6.4, can be used as a tool for analyzing the GP system proper. 'GP Cross' appears to be very similar to the transposition operator described in section 6.3 (hence the use of the term 'transposon-vector' which was used to describe the parse tree fragments exchanged during the simulation). The difference is that the transposons are exchanged between chromosomes, as opposed to within a chromosome. By viewing the GP crossover as a type of transposition between chromosomes, the composition of a GP chromosome can

be thought of as a massive number of transposons linked together, with transposition rearranging them at will.

The exchange of transposons between chromosomes is also used in nature. For example, it is the mechanism by which the multiple drug resistances developed in a population of bacteria. Resistance to one or two drugs may be developed through mutations in individual bacteria. Then, through transposition, these resistances can be exchanged, and combined, to produce a bacteria that has multiple drug resistance.

Viewing the GP system from within the EP-I framework also provides a good intuition as to why GP does not need mutation, when all other GA systems developed will not converge without it.

Koza, in his book 'Genetic Programming', correctly describes the use of mutation in a GA. Mutation reintroduces alleles, which may be important to the correct solution, but have been prematurely eliminated from the gene pool at a given locus. However, he dismisses mutation as being inapplicable to GP, since, as he correctly concludes, there are no fixed positions in the GP chromosome, and then refers to it as being only a minor operator, and so can be dispensed with. However, while mutation is only minor in its immediate effects during reproduction, it is absolutely vital to the convergence of a regular GA. Crossover and simple reproduction, in a regular GA, cannot add back the necessary alleles that may have been bred out at a given locus, or were, unfortunately, never produced there during initialization. Consequently the system will not converge. Yet GP does converge without mutation. Why?

The answer lies in the transpositional nature of the crossover. Unlike the normal GP system, where alleles and loci are ill-defined, or possibly undefinable, the GP chromosome simulated by EP-I does have specific loci with genes and their alleles. It becomes obvious that with the 'transposition like' crossover, it is impossible for an allele

to be eliminated in the gene pool at any locus. If, during a given generation, there are no values of an allele at a given locus in the population, then, by the next generation, as long as the value of that allele exist somewhere in the gene pool, it could be transported to that locus through the 'transpositional' crossover. Mutation is, therefore, not necessary to reinstate an allele to a locus; it could never be permanently lost from the locus in the first place.

It is possible that a gene could be lost from the entire gene pool. In other words, gene 'x' could be lost to the entire population, at all loci. If this happens the GP system, without the mutation operator would fail to converge (provided the gene is needed in the solution). However, the chance of that happening is remote, since the number of operators is small in comparison to the number of loci in the gene pool, and if it does, the system is just restarted as a new "niche", with the lost gene back in the population.

In this way, the benefits of mutation are seen to be incorporated implicitly, by the transposition effect, into the GP crossover. This is why mutation does not have to be used by GP.

8.3 CONCLUSION AND FUTURE WORK

8.3.1 CONCLUSION

The EP-I system has shown that an unmodified GA can, indeed, perform program induction. The GP crossover, while vital for the convergence of GP, does not have any special properties that are necessary to solve program induction problems. Furthermore, EP-I has lead to new insights into both GP, and program creation through evolution.

Insight into the ability of GP to converge to solutions, without the apparent use of a mutation operator, was provided by analyzing the GP system as simulated in EP-I. It was found that the beneficial effects of mutation are performed by the transpositional properties of the GP crossover.

EP-I by being a second system that uses the same primitives for evolving programs as GP, can also, through comparison, allow general observations on program evolution that were not previously perceived. For example, there seem to be inherent properties in the design of the primitives used for program evolution, which affect the rate of convergence, but are independent of the genetic manipulation methods used by the system. This was evidenced when the two systems, which use vastly different genetic operators, obtained similar convergence rates when trying to solve the same problems using the same primitives. This occurred for each problem in the test suit. While still only an anecdotal result at this point, it is intriguing, and worthy of further investigation.

Finally, EP-I introduced a new encoding feature, and a new genetic operator. An Intron is an adaptive, meta-level, gene mask, that can remove from consideration, unimportant gene segments, thus reducing the search space, and therefore increasing the convergence rate. Furthermore, introns are affected by the same selection pressures that are felt by normal genes, and so acquire the same evolutionary benefits. Transposition, although not technically a new genetic operator, has been separated from the concept of inversion, with which it had been previously confused, and identified as a useful method for distributing beneficial, modular, genetic code from one area of the chromosome to another. This alleviates the need to develop the same sequence independently at many different locations in the chromosome. Hopefully, both of these operators will be useful in other GA systems to increase convergence speeds, or maybe allow problems to be solved by the GA method that were not previously possible.

8.3.2 FUTURE WORK

Further experiments need to be performed to more fully explore the properties of the EP-I system. It is important to know how different tree heights affect the rate of convergence. Do the expanded parse trees have to be approximately the size of the optimal solution, or will the introns allow solutions to be quickly generated from much larger trees? The effects of the population size on convergence should also be studied. Since the EP-I system uses allelic replacement instead of tree swapping, and a smaller chromosome size, it may be possible to reduce the size of the population and still keep similar rates of convergence. Also, independent studies of the effects of large infrequent mutations should be performed on problems other than program induction, to see what effects they have, and what improvements can be made to systems in which they are used.

Once a good base line of experimental results have been developed for EP-I, the system can be used as a tool for studying the properties of introns, and the transposition operator, and their effects on the ability of the system to find solutions. Ultimately a theoretical basis for program induction in a GA environment is desirable. Hopefully the results obtained by EP-I will be helpful in pointing the way towards such a theory, and, once obtained, EP-I may again be useful by providing a tool to test the theory and its predictions.

REFERENCES

- [Antonisse 89] Antonisse, J., "A New Interpretation of Schema Notation that Overturns the Binary Encoding Constraint". In *Proceedings of the Third International Conference on Genetic Algorithms*, George Mason University, June 4-7, 1989, edited by J. D. Schaffer. San Mateo, California: Morgan Kaufmann Publishers, Inc., 86-91.
- [Baker 85] Baker, J. E., "Adaptive Selection Methods for Genetic Algorithms". In *Proceedings of an International Conference on Genetic Algorithms and their Applications*, Pittsburgh, July 24-26, 1985, edited by J. J. Grefenstette. Hillsdale, New Jersey: Lawrence Erlbaum Associates, 101-111.
- [Cramer 85] Cramer, N.L., "A Representation for the Adaptive Generation of Simple Sequential Programs". In *Proceedings of an International Conference on genetic Algorithms and their Applications*, Pittsburgh, July 24-26, 1985, edited by J.J. Grefenstette. Hillsdale, New Jersey: Lawrence Erlbaum Associates, 183-187.
- [Crick 79] Crick, F., "Split Genes and RNA Splicing" *Science*, Vol. 204 (April 20, 1979): 264-271.
- [Davis 91] Davis, Lawrence, *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold, 1991.
- [Dawkins 76] Dawkins, Richard, *The Selfish Gene*. Oxford: Oxford University Press, 1976.
- [De Jong 87] De Jong, K. A., "Genetic Algorithms: a 10 Year Perspective". In *Proceedings of the Second International Conference on Genetic Algorithms*, MIT, Cambridge, July 28-31, 1987, editor J.J. Grefenstette. Hillsdale, New Jersey: Lawrence Erlbaum Associates, 169-177.
- [De Jong 89] De Jong, K. A., "On Using Genetic Algorithms to Search Program Spaces". In *Proceedings of the Third International Conference on Genetic Algorithms*, George Mason University, June 4-7, 1989, edited by J. D. Schaffer. San Mateo, California: Morgan Kaufmann Publishers, Inc., 210-216.
- [Fogel, et. al. 66] Fogel, Lawrence J., Alvin J. Owens, and Michael J. Walsh, *Artificial Intelligence Through Simulated Evolution..* New York, London, Sydney: John Wiley & Sons, Inc., 1966.

- [Fujiki & Dickinson 87] Fujiki, C., and J. Dickinson, "Using the Genetic Algorithm to Generate LISP Source Code to Solve the Prisoner's Dilemma". In *Proceedings of the Second International Conference on Genetic Algorithms*, MIT, Cambridge, July 28-31, 1987, editor J.J. Grefenstette. Hillsdale, New Jersey: Lawrence Erlbaum Associates, 236-240.
- [Goldberg 89] Goldberg, David E., *Genetic Algorithms in Search, Optimization & Machine Learning*. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1989.
- [Grefenstette & Baker 89] Grefenstette, J. J., and J. E. Baker, "How Genetic Algorithms Work: A Critical Look at Implicit Parallelism". In *Proceedings of the Third International Conference on Genetic Algorithms*, George Mason University, June 4-7, 1989, edited by J. D. Schaffer. San Mateo, California: Morgan Kaufmann Publishers, Inc., 20-27.
- [Grefenstette 91] Grefenstette, J. J., "Lamarchian Learning in Multi-Agent Environments". In *Proceedings of the Fourth International Conference on Genetic Algorithms*, University of California, San Diego, July 13-16, 1991, edited by R. K. Belew, and L. B. Booker. San Mateo, California: Morgan Kaufmann Publishers, Inc., 303-310.
- [Hartl, et. al. 88] Hartl, D. L., D. Freifelder, L. A. Snyder, *Basic Genetics*. Boston: Jones and Bartlett Publishers, 1988.
- [Holland 75] Holland, John H., *Adaptation in Natural and Artificial Systems*. Cambridge, Massachusetts: MIT Press, 1992; first published University of Michigan, 1975.
- [Jefferson, Collins, et. al. 91] Jefferson, D., R. Collins, C. Cooper, M. Dyer, M. Flowers, R. Korf, C. Taylor, and A. Wang, "Evolution as a Theme in Artificial Life: The Genesys/Tracker System". In *Artificial Life II*, edited by Langton, Christopher, et. al.. Reading, Massachusetts: Addison Wesley, 1991.
- [Koza 90] Koza, John R., *Genetic Programming: A Paradigm for Genetically Breeding Populations of computer Programs to Solve Problems*. Stanford, California: Stanford University, Department of Computer Science Technical Report, Report No. STAN-CS-90-1314, 1990.

- [Koza 92] Koza, John R., *Genetic Programming: On the Programming of Computers by means of Natural Selection*. Cambridge Massachusetts: MIT Press, 1992.
- [Langley, et. al. 87] Langley, Pat, H. A. Simon, G. L. Bradshaw, and J. M. Zytkow, *Scientific Discovery: Computational Explorations of the Creative Process*. Cambridge Massachusetts: MIT Press, 1987.
- [Levenick 91] Levenick, J. R., "Inserting Introns Improves Genetic Algorithm Success Rate: Taking a Cue from Biology". In *Proceedings of the Fourth International Conference on Genetic Algorithms*, University of California, San Diego, July 13-16, 1991, edited by R. K. Belew, and L. B. Booker. San Mateo, California: Morgan Kaufmann Publishers, Inc., 123-127.
- [Mendenhall, et. al. 81] Mendenhall, W., R. L. Scheaffer, D. D. Wackerly, *Mathematical Statistics with Applications, Second Edition*. Boston: Duxbury Press, 1981.
- [Quinlan 86] Quinlan, J. R., "Introduction of Decision Trees" *Machine Learning*, 1 (1), 1986, 81-106.
- [Spears & De Jong 91] Spears, W. M., and K. A. De Jong, "On the Virtues of Parameterized Uniform Crossover". In *Proceedings of the Fourth International Conference on Genetic Algorithms*, University of California, San Diego, July 13-16, 1991, edited by R. K. Belew, and L. B. Booker. San Mateo, California: Morgan Kaufmann Publishers, Inc., 230-236.
- [Stansfield 91] Stansfield, William D., *Theory and Problems of Genetics, Third Edition*. Schaum's Outline Series. New York: McGraw-Hill, Inc., 1991.
- [Steele 90] Steele, Guy L. Jr., *COMMON LISP: The Language, Second Edition*. Bedford, Massachusetts: Digital Press, 1990.
- [Syswerda 89] Syswerda, Gilbert, "Uniform Crossover in Genetic Algorithms". In *Proceedings of the Third International Conference on Genetic Algorithms*, George Mason University, June 4-7, 1989, edited by J. D. Schaffer. San Mateo, California: Morgan Kaufmann Publishers, Inc., 2-9.

END

3 1 - 0 5 - 9 5

FIN