

**SIMULATION OF THE EVOLUTION OF
SINGLE CELLED ORGANISMS WITH
GENOME, METABOLISM AND TIME-VARYING
PHENOTYPE**

Paul Joseph Kennedy,
B.App.Sci. (Computing Science) (1st class Hons)

Submitted to satisfy the requirements for the degree of
Doctor of Philosophy

University of Technology, Sydney
August 1998

CERTIFICATE

I certify that this thesis has not already been submitted for any degree and is not being submitted as part of candidature for any other degree.

I also certify that the thesis has been written by me and that any help that I have received in preparing the thesis, and all sources used, have been acknowledged in this thesis.

Signature of Candidate

.....

dated

ACKNOWLEDGEMENTS

Many thanks are due to Dr. Tom Osborn for his guidance and insight during this project, particularly in its early days. Thanks also to Eugene Dubossarsky, Dr. Brian Lederer and Chris Trengove for their perceptive comments during our many caffeine-soaked sessions at the Froggoire and to Mustapha for his special *café au lait*. And last, but never least, thanks to Rick Welykochy and my parents for everything.

CONTENTS

CONTENTS	III
LIST OF ILLUSTRATIONS	IX
LIST OF TABLES	XI
ABSTRACT	XIII
CONTRIBUTIONS TO KNOWLEDGE	XV
CHAPTER 1 INTRODUCTION	1
LAYOUT OF THESIS	5
CHAPTER 2 LITERATURE REVIEW	6
2.1 CELL MODELS	6
<i>Rosenberg</i>	6
<i>Weinberg</i>	9
<i>Enzyme Networks and Automata</i>	10
<i>Fleischer and Barr</i>	13
<i>Kauffman Gene Switching Network</i>	13
<i>Neural Models</i>	14
<i>Wilson</i>	14
<i>Metaphors for Cell Models</i>	15
2.2 BIOLOGY	16
2.3 CHEMICAL KINETICS	17
2.4 GENETIC ALGORITHMS.....	19
<i>The Inversion Genetic Operator</i>	23
<i>The Broadcast Language</i>	24
2.5 OTHER RELEVANT AREAS	26
CHAPTER 3 OVERVIEW OF CELL MODEL	28
3.1 CELL MODEL	28
<i>Evolution of the Cell</i>	29
<i>Simulation of the Cell</i>	31
<i>The model as a coevolutionary system</i>	32
3.2 CHEMISTRY MODEL	34
3.3 CHEMICAL MODEL	35
<i>Chemical Type</i>	36
<i>Chemical Shape</i>	37

<i>Chemical Concentration</i>	37
CHAPTER 4 MODEL OF METABOLISM	39
4.1 BRIEF DESCRIPTION OF CELL METABOLISM	40
4.2 METABOLIC REACTIONS	41
<i>Adding New Reactions</i>	42
<i>Calculation of Rates for Enzyme Catalysed Reactions</i>	43
<i>Differential Equations for Metabolic Reactions</i>	44
4.3 PROTEIN DEGRADATION.....	47
CHAPTER 5 MODEL OF GENOME	49
5.1 GENOTYPE.....	49
5.2 DOUBLE STRAND BIT ENCODING	51
5.3 GENOME BIT ENCODING	53
<i>Gray Coding</i>	53
<i>Base Coding</i>	54
<i>Goldberg's Coding Principles</i>	56
5.4 BIOLOGICAL MOTIVATIONS	56
<i>Transcription</i>	57
<i>Translation</i>	57
<i>Gene Regulation</i>	58
5.5 THE GENE LANGUAGE	59
<i>Promoter Region</i>	60
<i>Genes</i>	61
5.6 CHARACTERIZATION OF THE NON-CODING REGIONS.....	69
5.7 GENETIC OPERATORS.....	69
<i>Inversion</i>	70
CHAPTER 6 GENE EXPRESSION AND REGULATION	73
6.1 ABSTRACTING GENE EXPRESSION AND REGULATION.....	74
6.2 FORMULATION OF GENE EXPRESSION AND REGULATION	74
<i>Spiders</i>	77
6.3 REACTIONS AND DIFFERENTIAL EQUATIONS REQUIRED FOR GENE EXPRESSION	78
<i>Gene Expression Reactions</i>	79
<i>Differential Equations</i>	81
6.4 GENE REGULATION	82
CHAPTER 7 MODEL OF INTERACTION WITH ENVIRONMENT	85
7.1 ENVIRONMENT	86
7.2 CELL GROWTH: MODIFYING THE CELL MEMBRANE.....	86
<i>Calculating the Size of the Cell</i>	87
<i>Effects of Changing the Size of the Cell</i>	88

<i>The Nature of Cell Wall Molecules</i>	89
7.3 TRANSMEMBRANE DIFFUSION.....	90
<i>Background Diffusion</i>	90
<i>Facilitated Diffusion</i>	92
CHAPTER 8 MODEL OF GENETIC ALGORITHM	96
8.1 THE GENETIC ALGORITHM SHELL.....	97
<i>Building a Phenotype from a Genotype</i>	97
<i>Fitness Function Used to Score Phenotypes</i>	100
Volume Metric	101
Cell Lifetime Metric	104
Switch Correlation Metric.....	104
Reaction Graph Complexity Metric.....	105
Membrane Modification Metric.....	105
Parsimony Metric.....	105
<i>Selection of Parents</i>	106
<i>Population Management and Replacement Strategies</i>	106
Replacement Strategy Used.....	107
Strategy 1: Replace Lowest Scoring Cell.....	107
Strategy 2: Replace Oldest Cell	108
Strategy 3: Alternate Between 1 and 2.....	108
8.2 EFFECTS OF RUNNING OVER A NETWORK OF PROCESSORS.....	109
8.3 PROGRAM VERIFICATION	110
CHAPTER 9 BOOTSTRAPPING PROBLEM	111
9.1 THE BOOTSTRAPPING PROBLEM.....	112
9.2 ENVIRONMENTS FOR THE CELLS	112
9.3 ENVIRONMENT A EXPERIMENT	114
<i>Case Studies of Individual Cells in Environment A</i>	120
9.4 ENVIRONMENT B EXPERIMENT	124
<i>Case Studies of Individual Cells in Environment B</i>	128
9.5 DISCUSSION.....	132
<i>What techniques did the cells discover to cope with the environments?</i>	132
<i>Weeding out the Shortsighted Solutions</i>	134
Changing the fitness of only the mother	139
Changing the fitness of only the father.....	141
Changing the fitness of both parents.....	143
<i>Effect of the Lamarckian Pressure</i>	145
CHAPTER 10 OTHER EXPERIMENTS.....	149
10.1 GENETIC OPERATORS.....	150

<i>Crossover</i>	150
<i>Mutation</i>	151
<i>Inversion</i>	153
10.2 FACILITATED DIFFUSION.....	154
10.3 COMPARING A BIT GENOME WITH A NIBBLE GENOME	157
10.4 COMPARING A SINGLE STRAND GENOME WITH A DOUBLE STRAND GENOME.....	161
CHAPTER 11 CONCLUSION	165
FUTURE RESEARCH	168
<i>Research with the Current Model</i>	168
<i>Extending the Model</i>	171
<i>Related Future Work</i>	172
GLOSSARY	174
APPENDIX A	181
APPENDIX B.....	185
B.1 CALCULATION OF THE WIDTH OF THE CELL MEMBRANE.....	185
B.2 CALCULATION OF THE VOLUME OF THE CELL.....	185
B.3 CALCULATION OF THE SURFACE AREA OF THE CELL	187
B.4 CALCULATION OF THE NUMBER OF WATER MOLECULES IN THE CELL	187
APPENDIX C	188
APPENDIX D	190
CLASS DIAGRAMS	190
RELATIONSHIP OF CLASSES TO THE BODY OF THE THESIS	190
PROGRAM SPECIFICATIONS	198
<i>GACell</i>	198
Population	198
Experiment.....	199
ProcessorPool.....	200
Processor	201
<i>CellSim</i>	202
Main Program	202
Environment and Cell classes	203
Vessel	203
ChemOut	205
World.....	205
Cell	206
Genome classes.....	210
Haploid.....	210

Chromosome	210
NibbleMixin	211
BitHaploid	212
DoubleStrandBitHaploid	212
DoubleStrandNibbleHaploid	213
CellChromo	214
ParsedOperon	216
Chemicals and Chemistry classes	217
Chemical	217
AbstractChemicalRep	219
OrdinaryChemicalRep	222
EnzymeRep	222
PartlyExpRep	223
CarrierRep	223
BoundCarrierRep	224
GeneSwitchRep	224
WaterRep	225
WallRep	225
Chemistry	226
Differential equation classes	226
CellDiffEqu	226
DERep	228
GeneUtil	229
GeneSwitchDERep	229
BagleyDERep	230
OrdChemDERep	231
EnzymeDERep	231
WaterDERep	231
WallDERep	232
PartlyExpDERep	233
CarrierDERep	233
BoundCarrierDERep	233
CellReactTerm	234
Reaction classes	235
Reaction	235
IdxReaction	236
TransReaction	236
EnzymeDegradationReaction	237
DiffusionReaction	238
Support classes (others are not given)	239

BucketArray	239
TwoDPos	239
TwoDBArr	240
REFERENCES	241

LIST OF ILLUSTRATIONS

Figure 2.1 How the genome sees its environment	25
Figure 4.1: Matching an enzyme against two substrates.	43
Figure 5.1 Representation of genome	52
Figure 5.2 Encoding scheme from Nature	53
Figure 5.3: Operons. a: structure of an operon, b: bases in an example of an operon	60
Figure 5.4: Deterministic Finite State Automata used to parse genome.....	64
Figure 5.5: Effect of gene ordering by inversion	71
Figure 5.6: Inversion algorithm	72
Figure 6.1 Discrete model of gene expression	75
Figure 6.2 Repressible operons. a: Without blocker, b: blocker is bound	76
Figure 6.3 Inducible operons. a: Without activator, b: activator is bound.....	77
Figure 6.4: Gene expression reaction graph	79
Figure 8.1: change in volume fitness component	103
Figure 8.2 Comparative Replacement Strategies	109
Figure 9.1 Fitness for environment A	116
Figure 9.2 Fitness by Mother's Fitness.....	117
Figure 9.3 Fitness by Father's Fitness	118
Figure 9.4 Change in Fitness after Rerun of Final Population	119
Figure 9.5 Fitness for Environment B.....	125
Figure 9.6 Comparison between Environments	125
Figure 9.8 Fitness by Father's Fitness	127
Figure 9.9 Change in Fitness After Rerun of Final Population	128
Figure 9.10 Change Fitness of Parents.....	135
Figure 9.11 Rerun change in fitness when changing neither parent	137
Figure 9.12 Rerun change in fitness when changing mother only.....	137
Figure 9.13 Rerun change in fitness when changing father only	138
Figure 9.14 Rerun change in fitness when changing both parents.....	138
Figure 9.15 Fixed Initial Conditions	146
Figure 9.16 Child fitness by mother's fitness for fixed 2 experiment	147
Figure 9.17 Child fitness by father's fitness for fixed 2 experiment	148

Figure 10.1 Vary Crossover Rates	151
Figure 10.2 Vary Mutation Rates.....	152
Figure 10.3 Vary Inversion Rates	154
Figure 10.4 Facilitated Diffusion	157
Figure 10.5 Double-stranded Bit Genome	160
Figure 10.6 Single Strand Genome vs Double Strand Genome.....	163
Figure B.1 cell membrane sphere packing.....	185
Figure D.1: Legend of chart notation.....	192
Figure D.2: Chart for Genetic Algorithm Driver Program	193
Figure D.3: Chart for Cell Simulation Program.....	194
Figure D.4: Chromosomes and Haploids	195
Figure D.5: Chemicals and ChemicalReps	196
Figure D.6: CellDiffEqus and DEReps.....	197

LIST OF TABLES

Table 3.1: Types of chemicals	36
Table 3.2: Properties of chemicals	37
Table 4.1: typical values for rate constants	45
Table 5.1 Base codings	55
Table 5.2: Context free grammar definition of the gene language. It deals with only one operon not an entire genome.	63
Table 5.3: Actions associated with each state change in the finite state machine.	65
Table 5.3 (cont'd): Actions associated with each state change in the finite state machine.	66
Table 5.3 (cont'd): Actions associated with each state change in the finite state machine.	67
Table 5.3 (cont'd): Actions associated with each state change in the finite state machine.	68
Table 5.4: Typical rates for genetic operators.....	70
Table 8.1: Number of instances of each kind of variable	98
Table 8.2: Differential equations and terms that apply to each variable.....	99
Table 9.1: Chemical and concentrations in environments A, B and parentless cells....	114
Table 9.2 Correlation coefficients for environment A.....	117
Table 9.3 Operons in first example cell.	121
Table 9.4 Membrane building and breaking molecules in the first example cell.	121
Table 9.5 Operons in second example cell.....	123
Table 9.6 Membrane building and breaking molecules in the second example cell....	124
Table 9.7 Correlation coefficients for environment B	126
Table 9.8 Operons in third example cell.	129
Table 9.9 Membrane building and breaking molecules in the third example cell.	129
Table 9.10 Operons in fourth example cell.....	131
Table 9.11 Membrane building and breaking molecules in the fourth example cell.	132
Table 9.12 Average values and standard deviations for the fitness of living cells in the final populations and their rerun values and the estimated number of dead cells in each population.	139

Table 9.13 Changes in sub-populations during the “Change Mother Only” experiment.	140
Table 9.14 Changes in sub-populations during the “Change Father Only” experiment.	142
Table 9.15 Changes in sub-populations during the “Change Both Parents” experiment.	144
Table 9.16 Correlation coefficients for fixed 2 experiment.....	148
Table 10.1 Difference in values for parameters between control experiment and facilitated diffusion experiment	155
Table 10.2 Comparative numbers of genomes and schemata for four different encoding schemes	159
Table 10.3 Parameters for the single-stranded and double-stranded genome experiments.	162

ABSTRACT

A novel model of a biological cell is presented. Primary features in the cell are a genome and metabolism. Pairs of genome and metabolism coevolve with a genetic algorithm (GA) to produce cells that can survive in simple environments. Evolution of the genome is Darwinian, whereas evolution of the metabolism has Lamarckian features through acquired chemical concentrations being inherited. Fitness is more closely correlated with the mother cell than with the father. A biologically inspired double-strand genome model is presented. Double-stranded genomes admit a large increase in the number of schemata represented by each genome compared to single-strand encodings. This gives GAs more information to use and allows faster search. Simple implementation of a biologically inspired algorithm for inversion also becomes possible, as well as a compression of data on the genome. Increased rates of inversion showed an increase in population convergence. Double-stranded genomes impose constraints between strands that decrease the overall rate of population convergence. Four-bit bases from a parallel genomic language are encoded on the genome. The parallel genomic language, following the operon model of Jacob and Monod, allows genes to be placed on the genome at any loci and allows easy implementation of an inversion operator. The genome and chemical metabolism of a cell in our model have a close relationship. Genomes specify allowable families of enzyme-catalysed chemical reactions and families of chemicals that may diffuse through the cell membrane at increased rate. Chemicals produced from metabolic processes regulate genes and allow expression of proteins from the genome. We introduce the “bootstrapping” problem: evolution of cells stable in simple environments from random genomes and initial simple metabolic conditions. Experiments show that solution of the “bootstrapping” problem is much easier with coevolution than when the initial metabolic conditions remain fixed. A gallery of cellular survival strategies is given. Genes in the population are diverse because there is a variety of equally valid solutions to the problem posed by the environment. Solution to the “bootstrapping” problem is hindered because fitness functions cannot differentiate between cells using myopic solutions rather than long-term strategies. Cells with myopic strategies attain high fitness but produce offspring with high probability of cell death (ie, when the myopic solution begins to fail). A novel

solution, where fitness of parents is retroactively modified when the fitness of offspring becomes known, reduces the number of cells exhibiting myopic strategies.

CONTRIBUTIONS TO KNOWLEDGE

The main focus of this thesis is to model the evolution of the control of a complex metabolism. We wished to determine and delineate factors in a complex genome controlled cellular metabolism. These were identified using a simulated cell constrained by cell size and subjected to varying environmental conditions.

1. Development of a novel model of a biological cell that combines a genome and chemical kinetic metabolism to produce a time-varying phenotype.
2. A feature of the model is that cells are coevolved pairs of genome and metabolism, each of which evolves in a different way: Darwinian and Lamarckian. This coevolution results in more rapid discovery of problem optimisation within the evolving population.
3. Identification and solution of a “bootstrapping” problem. That is, we found that it is possible to coevolve a genome and metabolism from random beginnings to produce a cell that acts as a coherent union and that can survive in a simulated environment.
4. Invention of a novel biologically inspired encoding scheme for genetic algorithms: a double-strand encoding.
5. This new encoding scheme makes possible a simple biologically inspired algorithm for the inversion genetic operator.
6. Double-strand encoding dramatically increases the number of schemata represented by one genome, therefore allowing more efficient search for genetic algorithms because of the increased information.
7. Development of a parallel genomic language that encodes high level semantic information on the genome so that genes are not fixed to particular loci.
8. Development of an algorithm for gene expression and regulation that translates strings written on the genome in the genomic language into terms in a system of differential equations.
9. In our model, cells encode problem-solving strategies. The fitness function is not able to determine the potency of each strategy before the cell breeds. We have developed a method for dealing with the uncertainty by modifying the fitness of

individuals retroactively. This method successfully eliminates cells with poor strategies from the population.

Chapter 1

INTRODUCTION

*Approach it as you would a love-affair:
You meet, you feel the spell, and linger there,
And by and by you think yourself enchanted.
At first you thrive; then obstacles are planted;
You walk on air, then fall to bittersweet:
And thus, behold, your romance is complete
- Johann Wolfgang Von Goethe, "Faust" Part I*

This chapter introduces our model of a biological cell and the way we evolve populations of them. We give our motivation for various aspects of the simulation and summarise results for experiments we performed on the cells.

In this work, we introduce a new model of biological cells inspired by Rosenberg's 1967 thesis "Simulation of Genetic Populations with Biochemical Properties". Our model is similar in spirit to Rosenberg's model but, given the advances in computing power during the last thirty years, it is possible for it to be much more elaborate. Rosenberg used a card-fed mainframe whereas we use networks of Sun workstations. Even with these tremendous increases of computing power, there are still severe limits on what can be achieved with a simulation of cells. Thus our simulation of a few thousand cells each having up to a hundred or so genes cannot compare with the tremendous computing power implicit in even a small backyard containing millions of blades of grass, each with millions of cells running on solar power for millennia of evolutionary time. How can we ever hope to simulate such a remarkable system as a single cell in any complexity? No wonder natural evolution has solved the problems posed by so many different, varied and harsh environments – sulphur springs, deep

underwater, high altitudes, dry deserts, etc. producing solutions such as flight, intelligence and sight!

Our simulation is a biological metaphor. It is more, though. It is also an abstract model that might be used to investigate topics in control, homeostasis and parallel programming. Not all aspects of a cell are modelled at the same depth. We modelled the parts of a cell that we thought were important in governing how a cell functions: the parts where control impinged. Cells are such complicated systems that it is not computationally feasible to model them faithfully in complete detail.

Each cell in our model contains two main components: a genome and a metabolism. The genome stores the genetic material of a cell, whereas the metabolism represents the chemical reactions that occur in the cell. These two components affect each other intimately. The genome controls the metabolic reactions that may take place in the cell and the metabolism forms a context for the genome. In this model, the metabolism and genome are passed to daughter cells in different ways. Initial metabolic conditions for daughter cells are derived solely from the maternal cell. There is a Lamarckian aspect to this evolution. The genome, on the other hand, is derived sexually from two parent cells. This is more Darwinian. Well adapted, functional cells have a metabolism and genome that complement each other and work synergistically. The main motivation for our work is to examine the relationship between the genome and metabolism and to determine the difficulty of evolving well-adapted pairs of genome and metabolism. In particular, we are interested in the sensitivity of the genome to different initial metabolic conditions. To model this evolutionary process, we use a genetic algorithm. The model of individual cells is encoded in a large system of differential equations that are solved using an adaptive Runge-Kutta numerical integration algorithm for a finite lifetime.

We are interested primarily in the possibility of evolving germ lines of cells to exist in simple abstract environments. The environment exerts a pressure on the cell it contains by allowing chemicals to diffuse into the cell. These chemicals cause the cell to grow or shrink in volume. We wish to evolve cells that can counter this pressure and maintain their size or grow slightly. The “bootstrapping” problem, then, is to evolve (from random genome and simple initial metabolic conditions) cells that can exist in their

environment. It transpires that many distinct solutions to the bootstrapping problem are possible and relatively simple to find. Cells with high fitness are found soon after breeding commences and it is difficult to improve upon these solutions.

The major difficulty with evolving solutions to the bootstrapping problem is to formulate a fitness function for the genetic algorithm. In some ways, this function is a measure of the 'life' exhibited by the cells. We strove to make this function as abstract from the particularities of the genome or the structure of the metabolism. Cells implicitly attack the problem posed by the environment using different strategies. Some strategies are shortsighted and, although they are superior to those of other cells, they may fail miserably for their descendants. The problem is that, as they are superior among other strategies of contemporary cells, they are selected by the genetic algorithm and passed on to offspring. The troublesome part of the fitness function is differentiating between these myopic strategies and the ones that succeed for longer terms. We argue that it is essentially impossible to devise a fitness function that can perform this task.

It is, however, possible to work around this deficiency in the fitness function. We allow the cell to produce offspring based on the best guess of fitness by the fitness function. Once the fitness of children is gauged, we retroactively change the fitness of the parent cells. The parent's fitness is increased if its child scored more or decreased if the child was not as fit. This scheme helps diminish the selection of the myopic solutions, although some shortsighted solutions remain. A scheme such as this is not as biologically plausible, but is required due to the very short time that we simulate cells compared to the time they live in real life.

The relationship between genome and metabolism reminds us of coevolution because each evolves separately. The fitness of child cells is more correlated with its mother cell than with its father cell. We were interested in the sensitivity of cellular evolution to initial metabolic conditions. To this end, we conducted experiments keeping the initial chemical concentrations the same for all cells. That is, stopping the mother cell from passing her modified chemical concentrations to her offspring. These experiments showed that evolution of solutions is improved when coevolution is permitted to occur.

We devised a novel double-stranded genome loosely based on the structure of DNA molecules. The double strands are distinct from diploidy. This haploid genome encodes four bit bases (nibbles) which are tokens in a genomic language based on the operon model of Jacob and Monod (in addition to molecular stability from hydrogen bonds between strands in the double helix structure of Nature). Although this is a context sensitive language, a finite state machine was devised to parse genomes into parse trees of operons. With the metabolism, these parse trees are translated into terms in the system of differential equations.

The double strands in the genome allow us to implement a novel biologically inspired algorithm for the inversion genetic operator. Experiments show that increased rates of inversion aid considerably in convergence of average fitness of a population of cells towards the maximum fitness.

We were motivated to determine whether a double-strand bit genome performed better than a double-strand nibble genome. By this, we mean whether constraining the inversion and crossover genetic operators to nibble boundaries hindered evolution. Although results were close, experiments showed that the bit genome (ie, operators not constrained to nibble boundaries) performed slightly worse than the nibble genome.

We were also motivated to conduct experiments comparing performance of the biologically inspired double-strand genome with the single-strand genome abstraction that is more usual with genetic algorithms. Initially the double-strand genome performs better than single stranded genomes. Convergence of the population is much faster with single-strand genomes, however. This is true regardless of whether the length of the single-strand genome is the same as the length of one of the double strands or double this length (that is, the total length of the double-strand genome). Initial superior performance of the double-strand genome is attributed to an increase in the number of schemata represented by double-strand genomes compared to single-strand encodings. This increase in number of schemata, however, has the cost that crossover and mutation have mysterious effects on the schemata. These additional constraints imposed by the double strands affect convergence of the population adversely.

Layout of Thesis

Chapter 2 surveys literature associated with this project. In chapter 3, we present an overview of the cell model. Chapter 4 concentrates on our model of the metabolic processes in the cell. In chapters 5 and 6, we describe the genetic encoding and our model of gene expression and regulation. Chapter 7 describes the cell's environment and deals with the way cells and their environment interact. In chapter 8 we discuss the genetic algorithm used to evolve a population of cells. Chapters 9 and 10 describe experiments run on the cell. Chapter 9 discusses a major problem that we are interested in: the 'bootstrapping' problem. Chapter 10 discusses other experiments performed on the cells. In chapter 11, we summarise the main results of the thesis. Appendix A derives formulae used for gene regulation. In Appendix B, we calculate features of the cell such as its volume and the number of water molecules it contains. Appendix C derives a result used in the fitness function of the genetic algorithm: an estimate of the growth of the cell due only to the environment. Finally, in Appendix D, we describe the computer programs used to simulate the cell and its evolution.

Chapter 2

LITERATURE REVIEW

In this chapter, we examine literature relevant to the artificial cell model. Firstly, we examine a variety of cell models including Rosenberg's 1967 thesis. Rosenberg is a significant predecessor of our work. Subsequently we review the current status of cell biology and chemical kinetics. A short description of genetic algorithms is distilled from major texts in the considerable literature. Chapter 8 identifies the differences between the standard genetic algorithm and our implementation, which more closely follows cellular genetics. Finally, we mention other texts used.

2.1 Cell Models

Many models of cells have been constructed by researchers. Since cells are extremely complicated systems, there are many different ways to view a cell. Often, different models highlight different aspects of a cell. Some models, such as cellular automata, concentrate on abstract cell assemblies and have very simple models of individual cells. Other models, including ours, are interested in the internal behaviour of a cell. These represent individual cells in more detail and usually do not attempt to combine cells into assemblies.

Rosenberg

In many ways, our work is a successor to that of Rosenberg (Rosenberg 1967, Holland 1992, Goldberg 1989). In his 1967 thesis, he simulates the evolution of a population of single celled individuals using a proto-genetic algorithm. The phenotype of each cell consists of chemical concentrations that change over time as the result of enzymatic

catalysis. Presence or absence of enzymes is under genetic control. Interaction between the population of cells and their environment is dynamic. The thesis also explores the relationship between the number of crossover positions on the rate of evolution.

Each cell in the population consists of a set of idealised chemicals and a diploid genome. The genome codes up to 20 genes and each gene has up to 16 alleles. Associated with each gene is a chemical reaction. Each allele for the gene represents a different rate constant for the reaction. In this model, enzymes are implied through this form of enzymatic catalysis, although not specifically manifested as chemical species. Because the genome is diploid, two alleles are valid for each gene. Actual rate constants used for the reaction are the sum of the rate constants for each allele. In this way, dominance is elegantly handled for when the homozygous and heterozygous conditions are indistinguishable phenotypically (Gardner, Simmons and Snustad 1991) (ie, so long as the recessive allele rate constant is much less than the dominant allele rate constant). Codominance, where both alleles are fully expressed in a heterozygote, and semidominance, the dominance is not complete, are not modelled well because the rate constants are additive (Gardner, Simmons and Snustad 1991). In these cases, the maximum of the two rate constants and their average, respectively are better models.

The chemical kinetics model used for each cell is discrete: time is measured in indivisible increments. Numbers of molecules of each chemical species are manipulated with difference equations. The difference equations also take into account permeation of molecules through a cell membrane.

Sexual mating in the population occurs at regular intervals. Selection of parents is either uniform (random) or stochastically weighted towards fit cells. The fitness function used to score cells is relatively simple. It is the inverse of a function of cell properties. Rosenberg used only one cell property. By cell property, Rosenberg means that a cell has a set of chemicals with concentration similar to some preset values. Cell properties are given by equation (2-1). The closer f_i is to 0; the closer an individual is to possessing property i .

$$f_i = \sum_{j \in C_i} (X_j - \bar{X}_j)^2 \quad (2-1)$$

where f_i is the i th ‘property’ of the cell,

C_i is the set of chemicals associated with property i ,

X_j is the actual concentration of chemical j and

\bar{X}_j is the preset concentration of chemical j .

The number of offspring produced by a mating is a function of cell properties. Initial chemicals for offspring are taken from the environment of the new cell. The amount of chemical taken from the environment can be similar to its amount in a parent cell, or near the average concentration of all chemicals in a parent, or some amount between these extremes.

After conducting and analysing many experiments, Rosenberg formed a variety of conclusions:

- The greater the number of crossover positions on the genome, the greater the scope for a population to explore its search space.
- Initialisation of concentrations in cells is important. Rosenberg realised that initialising concentrations to that of a parent’s concentrations at the time of breeding can stop evolutionary trends. This is particularly true for simple fitness functions, such as his, that measure the concentrations in the cell against arbitrary fixed values. He noticed that the different initial concentrations of chemicals for cells negate genetic differences between them. He solved this problem by setting initial concentrations only for the simpler building block chemicals. The approach we take, however, is to pass on almost all of the chemicals from one of the parents (the “mother”) but make the fitness function less specifically dependent on fixed values of concentrations – but rather on cell behaviour in a more general sense. We look for the cell, as an entire entity, to exhibit some behaviour or to solve some general problem. We evolve cells to stay the same size, or grow slightly, under environmental pressure to change in size. Rosenberg, instead, looked for cells that

produced certain chemicals to given concentrations. For the kind of behaviour we want displayed, we find that coevolution between initial chemical concentrations and genome is beneficial and probably essential.

- Rosenberg noticed that stochastic variation affects small populations greatly. He found that an increase in the number of offspring for a single mating and more stringent rules for choosing cells for removal alleviated this problem. He also noted that it is important to control the number of offspring produced in each generation. Too many offspring forces the removal of fit cells. We observed a similar effect in our simulation.

There are a number of important differences between Rosenberg's model and ours. Many of these arise from the thirty years difference in computing resources. Whereas Rosenberg uses a discrete time model of chemical kinetics, we have a continuous time model. As well, his enzymes do not appear in the model as chemicals. In our model, enzymes are chemical species like any other. This difference means that our model of gene expression is more realistic (ie, biologically inspired) and our genome significantly more complex than his. The model of a semipermeable membrane that we use (section 7.2) is more detailed than Rosenberg's model. This allows us to have a more abstract fitness function.

Weinberg

Weinberg (1970) constructed a computer simulation of a living cell (Weinberg 1970, Holland 1992, Goldberg 1989). The model describes and predicts the biochemical behaviour of an *Escherichia coli* cell adapting to three different environments. Each environment specifies a different growth medium. Phenotypic adaptation of the cell was by means of allosteric modification of enzymes. To build and validate the model, simulated results were compared to laboratory results from the literature.

The model consisted of a metabolic network incorporating a small number of chemical pools. Chemicals that behaved similarly were 'lumped' together into aggregate pools. Weinberg used the idea of homomorphisms to lump species. Homomorphisms and

pooling are also discussed in Zeigler (1980). Pools used include ATP, ADP, glucose, amino acids, nucleotides, mRNA, tRNA, ribosomes, DNA and proteins. The protein pool is divided into enzymes and initiators. Each enzyme catalyses the conversion of molecules in one pool to those in another. Initiators are used to control production of amino acids into DNA replication sites which, in turn, are used to replicate the DNA pool. The mRNA pool is divided into separate mRNA ‘species’ used, with the ribosome pool, to control production of a specific enzyme ‘species’. The state of the model at an instant is given by the concentrations in each pool.

Transitions between states are achieved using difference equations. These model chemical kinetics in an abstract manner (because of the lumping). Allosteric modification of enzymes is modelled using three ‘species’ for each enzyme, each with an associated (and different) catalysis rate constant. The three species represented the enzyme, the enzyme bound to one amino acid and the enzyme bound to two amino acids. Difference equations related enzyme states to one another. Other difference equations modelled the repression of mRNA production, self-replication of DNA under genetic controls and the permeability of the cell to chemical pools in the environment.

Unlike Weinberg, we have not directly modelled the allosteric modification of enzymes. Other parts of our model (bound complexes) take it into account. Direct modelling of allosteric modification implies that the 3-dimensional geometry of enzymes is also modelled. This would make our model computationally infeasible.

Weinberg also suggested a genetic evolution of his cell model, although he did not build it. Evolution would proceed according to a genetic algorithm (Holland 1992) with DNA containing rate constants, similar to Rosenberg (1967).

Enzyme Networks and Automata

Holcombe (Holcombe 1994, Paton 1993) and Marijuán (1994) build cell models consisting of enzyme networks. Marijuán states that it is possible to build an automaton to replicate the behaviour of any reaction network.

Holcombe (1994) suggests modelling a cell using a hierarchy of automata. Each level represents a different level of organisation in the cell. Successive layers represent higher levels of control. Automata at each level may act sequentially or in parallel. At the lowest level, rate equations are modelled with sequential automata. The next level models the enzyme control of the systems of rate equations. This level may be a parallel automaton such as a Petri net. Holcombe used a Petri net at this level. The next layer models the way the enzyme control system is controlled. That is, DNA transcription and RNA synthesis and translation. Holcombe suggests that a final level may represent connections between organelles. Holcombe uses Eilenberg's X-machine, a generalised automaton, to model both sequential and parallel automata.

Two strengths are apparent in this hierarchy of X-machines. Holcombe (1994) and Marijuán (1994) note that X-machines are general computing machines with the same capabilities as Turing machines. This suggests that both Rosenberg's model and our model can be rewritten using X-machines. Paton (1993) adds that the hierarchy has the potential to model massively parallel processes.

However, Marijuán (1994) also lists five intracellular characteristics of cells that are not modelled well using automata. Some of these factors are not handled by any model. Marijuán lists these factors as “first of all, the formation of aggregates and clusters ... ; second, the complex role that water plays; third, the aging, oxidation and degradation problems in enzymes and proteins (and their turnover); fourth, the wondrous complexity that emerges in the dynamics of the collective networks; fifth, the special character that the logical function of the enzyme shows at the subatomic dimension”. Marijuán says that automata models are deficient in handling the first four factors and fail with the last. We model the first, third and fourth factors to some extent but also omit the fifth characteristic.

Holcombe makes the following statement in (Holcombe 1994):

“... we still maintain that there is a need for a hierarchy of machine-type models to describe various aspects of these complex biochemical systems. It is clear from software engineering, where *systems of a similar order of complexity* have to be described and

analysed, that a number of different models are required, each capturing a particular ‘view’ of the system at a suitable level of abstraction.” (my italics)

This statement would appear to be a gross underestimate of the complexity of biological cells.

Marijuán (1994) lists three properties of enzyme networks: self-organisation, self-reshaping and self-modification. By self-organisation, Marijuán means that a system of feedbacks to enzymes in the enzyme network causes the entire network to become stable. The self-reshaping property refers to a cell’s ability to modify its enzymes to significantly change the connectivity of the enzyme network. This modification is persistent but reversible. This allows a cell to respond in significant ways to signals. By self-modification, Marijuán refers to a cell changing its ensemble of enzymes and proteins by DNA and RNA. Self-organisation and self-modification feature in our model. Reactions in our metabolism model may compete for enzymes. This results in a form of self-organisation. Self-modification results from gene regulation. It is not possible to modify enzymes in our model, so the self-reshaping property cannot be exhibited.

Marijuán (1994) also remarks that cells have an unusual style of computation. Because proteins are continually being replaced and destroyed by the cell, there is a fundamental constraint on the rate of protein synthesis. This means that a cell must employ a parsimonious control structure. It cannot control everything in an environment, but must react to a limited number of stimuli. Marijuán notes that this traps cells in particular environments. The fundamental constraint on protein synthesis also occurs in our cell model.

Furthermore, their models exploit specific hierarchical organisation that they prescribe (ie, design), rather than facilitate as emergent organisation.

Fleischer and Barr

Fleischer and Barr (1994) have constructed a cell model and simulation testbed. The emphasis is on multicellular development and their motivation is to grow artificial neural networks.

The cell model consists of a system of differential equations, some of which model continuous behaviour in a cell (diffusion, movement, emission of a chemical species) and others that model discrete ‘events’ in a cell’s life (cell division, death).

Fleischer and Barr combine four developmental mechanisms into their model; chemical factors (diffusion); mechanical factors (movement, cell-cell adhesion); genetic factors (protein production); and electrical activity. The strength of the model lies in this richness of mechanisms. Cells in this model do not have a metabolism, but may react to stimuli through their differential equations (cell behaviour functions). These are hard coded into the system. As well, the genome is modelled as a system of differential equations that produce proteins at a given rate (cell state equations). Protein production may be regulated by changing variables in the system of differential equations.

Kauffman Gene Switching Network

Kauffman (1993) presents an extreme abstract model of a cell. He is interested in the genomic regulatory system of a cell and its relationship to ontogeny. The products produced by genes are not considered. He examines the ensemble of Boolean gene switching networks. That is, networks of genes regulating other genes where each gene is either on or off with no intermediate state. Kauffman is interested only in networks where the Boolean functions regulating each gene are “canalyzing” functions. These are defined as functions where at least one of the input variables has one value that alone is sufficient in determining the output value. As the number of inputs to a Boolean function increase, the number of functions that are canalyzing decreases. Consequently, Kauffman’s networks are sparsely connected (genes are regulated directly by few other

genes). Kauffman states that Nature seems to use only canalizing functions for regulating genes.

Kauffman finds parallels between the ensemble of Boolean gene switching networks and Nature. He describes a cell type as a recurrent pattern of gene activity (ie, an attractor) and finds that the relationship between the number of genes and the number of cell types is similar to that observed in Nature. Likewise, the Boolean gene switching networks can explain the distribution of cell-cycle times and the homeostatic stability of cell types. The ‘expected’ genomic network, exhibiting many feedback loops rather than a strict hierarchy, matches biology. The Boolean gene switching networks also explain the fact that cell types differentiate into only a few other cell types and that ontogeny occurs using branching pathways of differentiation. Boolean gene switching networks show that cell types share many expressed genes and differ only by a small fraction of the genes in the genome. The effect of selection on genomic networks is also assessed. Kauffman argues that the order implicit in genomic networks matches biological genomic networks and that selection’s role is to tune the detailed structure of the network rather than to supply the coarse order.

Neural Models

Much work on cell modelling has concentrated on neural models (for example Fleischer and Barr 1994, Hertz, Krogh and Palmer 1991, Rumelhart, McClelland and the PDP Research Group 1986). Most of these models are interested in ensembles of neurons and, thus, simplify the intracellular behaviour vastly. The McCulloch-Pitts model of a neuron, for example, fires when the weighted sum of its inputs reaches or exceeds some threshold. The internal machinations of the neuron are reduced to a single simple transfer function and time delays.

Wilson

Wilson (1989) described a proposed model for the evolution of simple multi-celled organisms. Production rules determining the differentiation of cells are encoded on a

genome. A genetic algorithm evolves a population of genomes. The model of individual cells is very simple. Cells are represented only as cell types: an alphabetic letter. Each production rule takes a cell type and context as its condition. For example, a rule may fire if the cell is of type A, and there is a cell of type B directly next to it. Contexts are null, an adjacency relation (as in the example) or a signal from other cells. The action part of each production rule specifies the cell types that replace the cell for which the rule fires. A null cell type in the action means that the cell is removed, one cell type in the action part specifies the cell changes type, and two cell types specify that the cell splits into two daughter cells. Associated with each production rule (and encoded on the genome) is a weight. The weight of each production rule is used to choose a rule when the conditions of more than one rule applies to the cell.

Metaphors for Cell Models

Paton (1993) lists five metaphors for describing cell models. Individual models may be viewed from more than one metaphor at the same time.

The first of these is the machine metaphor. Here, the cell is viewed in terms of finite state automata. Holcombe's automata hierarchy and Marijuán's enzyme networks both fit into the machine metaphor. Paton says that the benefit of the machine metaphor is that it provides a mathematical formalism for describing cells.

Next is the circuit metaphor. Here we are interested in the flow of information through networks. There may be a close relationship between the circuit metaphor and the machine metaphor. Automata may be used to specify a model, but it is usually viewed as a network. Paton makes the point that the circuit metaphor has traditionally been important to the life sciences, giving examples such as the Krebs's cycle and blood circulation. The models of autocatalytic networks discussed in section 2.3 by Farmer, Kauffman, and Packard (1986), Bagley and Farmer (1991) and Bagley, Farmer and Fontana (1991) are also good examples of the circuit metaphor.

The society metaphor views a cell as a computational ecosystem populated by autonomous agents (enzymes and other proteins). Local interactions between these agents result in global non-linear behaviour.

Next is the text metaphor. Paton says that this metaphor adds concepts such as context and interpretation. This metaphor relates well to our model. We view the metabolism as a context for interpretation of the genome. A genome and metabolism are necessarily closely coupled. The idea of ‘cellular information’ is related to this metaphor. Paton makes the point that the more a cell is broken into its constituent molecules, the less meaning there is in the cell. Order relates to meaning (and behaviour). In our model, some chemical species mean more to the cell than others do. This means that chemical species contain information that is useful to other parts and roles of the cell.

The fifth metaphor Paton uses is the spatial metaphor. Here, he makes the point that operations in a cell are often limited to particular regions of the cell. For example, DNA transcription only occurs in the nucleus. We have not modelled this spatial aspect of cells in our model presuming “well stirred” chemical kinetics as a simplification.

2.2 Biology

The cellular biology and genetics used in our model is standard knowledge taught to undergraduate students and largely accepted among researchers. This means that we were able to consult standard texts to build the model. Solomon, Berg, Martin and Vilee (1993) is a standard university textbook describing biology in overview. Detailed information about cellular biology was found in Alberts, Bray et al. (1994) and to answer particular questions about genetics we looked to Gardner, Simmons and Snustad (1991). Since our model is quite coarse and abstract with respect to the biology that it models, these three texts were considered sufficient for all biological questions we needed to answer.

Biology and genetics are very different disciplines to computer science. In the main, computer science is black and white and most information can be known exactly (discounting noncomputable problems, of course). Usually we can say for certainty

whether a model exhibits a particular behaviour. We can dissect the model down to individual bits and see every element quickly and easily. Experiments can be reproduced exactly. These attributes, though, are strangers to biology. The whole field is in shades of grey. Dissection of an experiment can destroy it and even then, its components must be inferred since they may be too tiny to see through a microscope. It often seems that evolution has taken advantage of any situation it can. When one asks a question, from a computer science perspective, expecting to get one answer, one often seems to find that biology does something in many different ways. Sometimes it is not possible to find an exact answer to a question. This means that we, computer scientists, must remain flexible in our thinking and modelling of biology.

Some existing mathematical models of biology were incorporated into our model. These include the ‘operon’ model of gene regulation (Gardner, Simmons and Snustad 1991, Alberts, Bray et al. 1994), Segel’s model of molecular diffusion (Segel 1980b) and Rubino’s (1980) model of facilitated diffusion.

2.3 Chemical Kinetics

Underlying the model of the cell is a model of chemical kinetics and a general model of chemistry. All actions made by the cell are accomplished using chemical reactions. Much work in analytical chemistry focuses on modelling chemical reactions. We found three main texts useful. Avery (1974) and Segel (1980a) present a basic explanation of chemical kinetics. Chaplin and Bucke (1990) concentrate solely on enzyme catalysed reactions. In our simulation, we are interested mainly in enzyme catalysed reactions. One problem with modelling any sizeable system of enzyme catalysed reactions is that of enzyme saturation. A clever solution to this problem was proposed by Farmer, Kauffman, and Packard (1986). Their work has a different focus than ours. They seek to understand the origin of life using autocatalytic reaction networks.

Models of autocatalytic networks proceeded from work on random graphs of chemical reactions (see Kauffman 1993 for example). Kauffman demonstrates that as longer polymers are added to a system, the number of reactions involving the polymers increases faster than the number of polymers. This means that once the system reaches a

certain complexity of polymers, there will be many ways to build each polymer, and autocatalytic networks will appear. It is important to stress that the random graph models did not contain any consideration of chemical kinetics. As such, they model an 'ideal' situation.

In Farmer, Kauffman and Packard (1986) a model of chemical kinetics is added to the random network model. This work is also reported in Kauffman (1993). They model a stirred vessel of chemicals. A food set of chemicals is pumped into the vessel at constant rate and the vessel overflows. The simulation uses a system of differential equations. Like the differential equations in our model, this system is dynamic as new chemicals appear that are combinations of older chemicals. In this model a chemical species can act as substrate, product and/or catalyst. It is therefore difficult to determine whether a species is saturated or unsaturated. Farmer, Kauffman and Packard solve this problem using an approximation that the binding velocity is the same for all intermediate species. We use this approximation in our model. In the model by Farmer, Kauffman and Packard, catalysis of a reaction is assigned to any species in the system (regardless of its shape) with a small probability P . Results of this simulation demonstrate that when P is smaller than a critical value the system does not exhibit autocatalysis and the growth rate of the reaction graph decays exponentially. When P is greater than the critical value, autocatalysis almost surely occurs and the reaction graph grows quickly. Growth, however, is limited because there is finite mass in the system.

Bagley and Farmer (1991) extend the previous simulation by adding a matching chemistry such that the catalytic ability of a species is determined from its shape rather than by an arbitrary probability. Behaviour under this scheme is similar to the previous simulation. They found under suitable conditions, a reaction network can focus most of the mass from its environment into relatively few species. For this to happen, the reaction network must be drawn from equilibrium (by pumping a food set into the vessel or other means), and polymerisation must be favoured (by appropriate rate constants). The main focus of this work was to compare such autocatalytic networks to simple metabolisms.

In Bagley, Farmer and Fontana (1991) spontaneous reactions (that is, uncatalysed ones) are added to the simulation. The researchers found that the autocatalytic reaction

networks exhibit a succession of 'punctuated equilibria'. The reaction network would settle into a fixed point. After some time, however, it would 'amplify' a spontaneous reaction and the reaction network would change to accommodate the new reaction. They draw parallels between the evolution of an autocatalytic reaction network and the evolution of biological organisms.

Other work modelling the human immune system (Farmer, Packard and Perelson 1986) is also relevant. This research describes a string matching algorithm similar to the one we use in the metabolic reaction model. The equations of motion of the immune system simulation are, like ours, dynamic. The differential equations, themselves, are also similar to ours.

On a slightly different note, Banzhaf, Dittrich and Rauhe (1996) shows how chemical reaction networks can be used to calculate functions. With this in mind, we can view the cell as a sort of computer, programmed in parallel with the genome: the machine metaphor.

2.4 Genetic Algorithms

We use a specialised version of the genetic algorithm optimisation technique to search for cells that perform well in their environment. There is a vast body of literature associated with genetic algorithms and their surrounding field (evolutionary computation). We base our discussion mainly on two seminal texts. "Adaptation in Natural and Artificial Systems" (Holland first edition 1975, revised 1992) was the first book describing genetic algorithms, written by the father of the field, John H. Holland. Goldberg's "Genetic Algorithms in Search, Optimization and Machine Learning" (1989) was the first textbook.

Genetic algorithms are an optimisation technique suggested by the natural selection of biological organisms. They quickly search through many potential solutions to a problem to find the best within a local region. Although not guaranteed to find the optimal solution, genetic algorithms, for complex problems, find progressively better solutions much faster than random or general systematic search.

Goldberg summarises how genetic algorithms differ from other optimisation techniques and search algorithms. Genetic algorithms, in general, are more robust. He attributes this to four characteristics. Unlike other optimisation techniques, genetic algorithms modify codings of parameters of the problem rather than the parameters themselves. This avoids problems such as discontinuities in a parameter. Genetic algorithms have an inherent parallel nature. They search using a group of points rather than one point. This allows the optimisation algorithm to avoid becoming stranded on local maxima. Genetic algorithms use a fitness score (payoff information, objective function value) to rate the value of a potential solution. This value is not domain specific. Because of this, genetic algorithms can be used on a wide variety of problems, particularly those where no methods have been specially developed. Genetic algorithms, though, tend not to perform as well as custom-built methods that use domain specific information. The final characteristic Goldberg lists is that genetic algorithms are probabilistic algorithms rather than deterministic. Genetic algorithms use random selection and combination to guide their search into areas that appear to be more optimal.

Genetic algorithms maintain a population of potential solutions whose parameters are encoded on binary strings (called genomes or genotypes). Each member of the population is scored using a fitness function. Scoring occurs using the member's phenotype. This is the physical realisation of the genotype. That is, the parameters of the solution implied by the coding. Usually, there is little difference between the genotype and phenotype. A new population is created by selecting pairs of parents from the previous population and breeding them to produce offspring. Parents are selected based on their fitness score. The higher their fitness, the more likely it is that the member of the population will be selected for breeding. Breeding entails the combination (called recombination or crossover) of the genomes of the parents to produce a child string. Because the child's binary string is a combination of the parents, it has attributes of each parent's solution. Occasionally, to replace lost alleles in the population, bits in the child's string are randomly flipped (mutation). Crossover and mutation, then, are genetic operators applied to genomes. Other genetic operators are possible. The population of offspring are, in turn, scored with the fitness function and bred to produce another population. This process continues until a solution is discovered with high enough fitness.

The standard algorithm can be summarised with the following pseudo code. Bold words imply variable names. Italicised words are comments.

```

Make a population of random genomes
while a good enough solution has not yet been found
    Score the population
    Build phenotypes from the genotypes in the population.
    Score each phenotype using the fitness function.

    Build a new population
    Make a new empty population: newPopulation.
    for each new offspring to breed
        Select two parents from the population.
        Crossover the parents to make a child.
        Perhaps mutate the child.
        Put the child in the newPopulation.
    end for
    population ← newPopulation
end while

```

The most common method of selecting parents from a population is roulette wheel selection. With this method, the likelihood of selecting an individual is proportional to its fitness. We use this method in our simulation.

Different methods of handling the population are also available. The usual way, as described above, is to generate an entirely new population for each generation. This gives rise to discrete generations. Another method was employed in Rosenberg (1967). In his early simulation, one population was maintained. When offspring were produced, they replaced existing members of the population. There are no discrete generations with this scheme. The generations blend, with grandparents, parents and children all cohabiting the population. With this scheme, an heuristic must be developed to choose the population member to replace with offspring. This latter scheme may be elitist: fit individuals remain in the population with their children.

Genetic algorithms search using schemata (Holland 1992, Goldberg 1989, Michalwicz 1996). Schematas compare genomes over the search space. Each schema belongs to the set $\prod_{i=1}^l \{V_i \cup \{*\}\}$ where l is the length of the genome, V_i is the set of values for the i th detector (gene/base) on the genome and $*$ represents a “don’t care” value. That is, any value V_i in V could be in that position. For a binary genome of length 5, $1**10$ and $***0*$ are two possible schemata. The length of a scheme is defined as the index of its first defined position subtracted from the last defined position. For example, the schema $1**10$ has length $5-1 = 4$ and the schema $***0*$ has length $4-4 = 0$. The order of a schema is its number of defined positions. The order of each example schema above is 3 and 1 respectively.

A single genome is an instance of many schemata. For example, a binary genome of length l is an instance of 3^l schemata: the value at each position of the schemata may be 0, 1 or *. When a genome is tested, many schemata are tested in parallel.

Reproduction assigns more trials to schemata that are fitter than the average and less to weaker schemata. Exponentially increasing numbers of trials are allocated to above average schema and exponentially decreasing numbers to below average schema. This operation occurs in parallel for each schema represented in the population of genomes.

Crossover makes new instances of schemata already in the population and generates novel schemata that have not occurred yet. Each crossover affects many schemata simultaneously. Holland called this parallel effect on schemata from changes in the population: intrinsic parallelism. Crossover disrupts shorter schemata less often than longer schemata. This means that short fit schemata proliferate in the population and can be used as building blocks toward a solution.

The inversion operator, when used, as in our work, permutes schemata without changing their fitness. Similarly to crossover, one inversion affects many schemata simultaneously. This is another example of intrinsic parallelism.

Together reproduction, crossover, inversion and mutation allow the genetic algorithm to discover over time fit, short, low order schemata.

The Inversion Genetic Operator

The inversion genetic operator has a chequered past in genetic algorithms research. Goldberg presents its history. Holland describes it in the original formulation of the genetic algorithm (Holland 1992) but, over time, it seems to have disappeared from common usage. Inversion changes genome linkage. It moves genes together that were far apart. In other words, it permutes the genome. For this to work, the semantics of a gene must not be associated with its position (locus) on the genome. In standard formulations of genetic algorithms, this is not the case. This means that inversion requires a special genome representation. In Holland's algorithm, the crossover operator must also be constrained to function only on homologous genomes. That is, genomes with the same gene order. Goldberg reports that early research on inversion by Bagley (1967) suggests that runs tended to be longer with inversion because the constraint on the crossover operator effectively forced a much lower crossover rate. The conclusion reached was that the problem was not epistatic (nonlinear) enough for inversion to be worth the cost. Goldberg recounts that in later work by Cavicchio (1970), where a different encoding scheme allowed unrestricted crossover with inversion, relatively high rates of crossover and inversion produced good results. Goldberg says that later work by Frantz (1972) which used many variations of inversion and crossover produced "no clear advantage for inversion in any form". Goldberg points out, though, that the problems were sufficiently nonlinear enough to see an advantage. This work appeared to ring the death knell for inversion for some time. The next work with inversion was involved with solving the travelling salesman problem by combining the crossover and inversion operators into one operator. This work does not interest us here.

There appear to be two main difficulties to be overcome when using the inversion operator successfully. The problem must be sufficiently nonlinear enough to make the effort worthwhile. Secondly, the genome must be encoded in such a way as not to incur the constrained crossover. That is, the semantics of genes must not be dependent on their position and the problem of repeated or nonexistent genes must be solved.

Repeated genes or nonexistent genes are the result of crossover and inversion and are the reason that the crossover operator is constrained to homologous genomes.

Mitchell and Forrest (1995) say that inversion is not often used “at least partially because of the implementation expense for most representations”. They call for more research in the inversion operator and in innovative representation schemes. Both areas are addressed in this thesis.

As an aside, Mitchell and Forrest (1995) also discuss some research on interactions between evolution and learning and the “Baldwin” effect. They remark that “other researchers are investigating the benefits of adding ‘Lamarckian’ learning to the GA and have found in some cases that it leads to significant improvements in GA performance”. Ackley and Littman (1993) are one of these researchers. We have observed the same effect (the latter part of section 9.5).

The Broadcast Language

Holland (1992) describes a language called the broadcast language that was devised to be encoded on genomes. The broadcast language allows the genetic algorithm to adapt representations of the environment to find correlations between features in the environment and performance. There are similarities between the broadcast language and our genome language.

Four characteristics were deemed necessary in the broadcast language:

1. The language must be string based for compatibility with the genetic operators and to allow the genetic algorithm to search with schemata.
2. Functional ‘units’ (the equivalent to procedures) should not be position dependent. This is so that the inversion operator can function.
3. There should be few symbols at each position on the genome so that a richer set of schemata results. The broadcast language has a 10 symbol alphabet.
4. The language should be complete.

The broadcast language consists of a set of production rules on the genome. Symbols, read from the environment, trigger the production rules and broadcast other symbols into the memory. The symbols read from the environment constitute information in the environment that is not strictly for payoff purposes and can therefore be exploited. Some rules may create new rules. In many ways, the broadcast language is similar to Prolog.

Since it is possible to model operons (see section 5.1) using the broadcast language, it is a superset of our genome language.

Interaction between the broadcast language and its environment is discrete. This begs the question, what is the genome language's environment? Is it the cell's metabolism, or is it the cell's environment? The genome is affected by molecules that regulate and express genes. These molecules come from two sources: production by the cell metabolism or diffusion in from the environment. The genome cannot differentiate chemicals from these sources. From the point of view of the genome, *its* environment is the metabolism (figure 2.1). Interaction between the genome language and its environment is continuous.

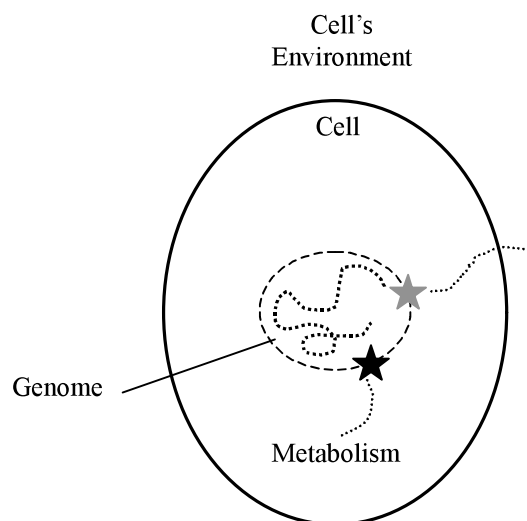


Figure 2.1 How the genome sees its environment

The broadcast language can rewrite itself as it runs. Although, in general, new rules cannot be added to the genome language as it runs, rules may be turned on or off. This leads to a similar result. Both languages allow genomes to modify an internal representation of their environment.

2.5 Other Relevant Areas

The encoding of genes on an artificial genome is more complex in our model than in previous genetic algorithm work. Encodings formed strings in a small custom-built language. Parsing of this genome language is a task eminently suited to computers since some of the most beautiful and profound proofs in computer science are associated with language parsing. Two standard texts for the theory of computation were routinely used (Brookshear 1989, Lewis and Papadimitriou 1981) as well as the famous “red dragon” book for compiler writing (Aho, Sethi and Ullman 1986).

At the heart of the cell simulation is a system of non linear differential equations. These were solved numerically with a computer program. The Runge-Kutta algorithm used was adapted from Press et al. (1995).

Designing and programming a model of this complexity is a difficult task, especially when the process is an evolutionary one itself. The development of the object-oriented programming paradigm in the 1980s allows software architects to build complex applications from small concrete abstractions. Consequently we designed our model using object-oriented analysis and design techniques and programmed it using the C++ programming language. The actual platform used was a Sparc 4 Unix workstation running the G++ compiler. Our simulations were run distributed over networks of around twenty workstations per experiment.

We recommend Stroustrup (1991) as a programming and reference text for the C++ language. Stroustrup also contains many useful analysis and design techniques. Coplien (1994) presents a variety of more advanced design constructs. Envelope and letterbox classes and reference counting are two such design idioms that proved to be particularly useful in the production of an efficient program.

Booch (1994) presents a notation for documenting the object-oriented design of a program. We have adopted his notation to give a specification of the cell simulation program (Appendix D).

Chapter 3

OVERVIEW OF CELL MODEL

God could cause us considerable embarrassment by revealing all the secrets of nature to us: we should not know what to do for sheer apathy and boredom.

- Johann Wolfgang Von Goethe, Memoirs (Riemer)

In this chapter, we describe the model of a single cell in the large. Later chapters explicate each part of the model, then show how the cells evolve. We describe how a cell is simulated using a system of differential equations and define a number of chemical species, each having particular properties.

3.1 Cell Model

Our model is an abstraction of a biological cell. Eventually they evolve together in a population, but here we will mainly outline the model of a single cell.

We assume the cell to be a single “well-stirred” vessel containing no organelles. Although we believe this assumption is reasonable, it is equally possible to view our cell model as an organelle model, in which case, the assumption becomes stronger. As well, we assume a simple environment for the cell. The environment consists of a very large “well-stirred” vessel consisting of a small number of simple chemical species held at fixed concentration. No reactions occur in the environment.

Each cell can be divided broadly into two separate, but tightly coupled, subsystems: a genome, containing hereditary control information, and a metabolism.

The *genome* specifies a set of protein molecules (enzyme or carrier molecules) able to be produced in the cell. Enzymes allow only certain reactions to occur in the metabolism and carriers allow accelerated selective diffusion between the cell and environment. Indirectly, through these molecules, the genome controls the metabolism and the cell. The genome also has finer control on the cell. It specifies the relative time required between the regulation of an operon and the change in rate of production of a protein in that operon (as the result of the regulation). Operons may consist of a number of genes, with each gene coding one protein. In this model, the time between regulation and action for genes that are close to the start of the operon is less than that of genes that are near the end of the operon. The genome consists of a sequence of bases. When read, in sequence, the bases encode sentences in a gene language.

Five main processes are embodied in the *metabolism*: modification of chemicals in the cell with enzyme catalysed reactions; expression and regulation of genes; growth of the cell; diffusion of chemicals through the cell membrane and protein degradation. The process of gene expression and regulation requires attention. Gene expression is the way that the metabolism creates enzyme and carrier molecules. Gene regulation allows the metabolism to switch genes on or off: causing the proteins concerned to be produced at different rates or not at all. In these ways, the metabolism controls the expression of the genome, and therefore itself and the cell.

Evolution of the Cell

The relationship between genome and metabolism can be quite complex and evolving each of these components using separate mechanisms adds to the complexity.

A cell's genome is a combination of the genomes of two parent cells, whilst the *initial* concentrations of a cell are usually taken as the *final* concentrations of the mother cell. This quality of the initial concentrations differs slightly from Nature. In prokaryotes, the mother cell divides giving some of her chemicals to the child cell, and in eukaryotes child cells start from special germ cells in the mother (Alberts, Bray et al. 1994).

This evolution of the components using separate means adds a coevolutionary aspect to the model. (Kauffman 1993). Evolution of the genome is Darwinian but, because a child cell inherits the final concentrations of chemicals in the mother cell, evolution of the concentrations has a Lamarckian component. Changes made to the mother's phenotype during her life are passed on to her children. However, we do not suggest that evolution of biological cells is Lamarckian. Although different evolutionary processes drive each subsystem, the complete cell must combine both into one synergistic mix.

The genome evolves a control structure, whereas the initial chemical concentrations form a context for the control. Chemical species can react together only in ways the genome specifies. This is the primary method by which the genome controls the cell. We say that the initial chemical concentrations form a context for control because genes on the genome are regulated by changing concentrations of chemical species. This gene switching alters elements of the control system.

Because the metabolism and genotype are so tightly coupled, it is clear that they must be well adapted to one another. A genotype for one strain of cells may not function properly with the metabolism from another significantly different family of cells. An analogy can be found in the control of a company, where the chief executive officer (CEO) represents the genome and the company represents a cell. Imagine placing an English (only) speaking CEO into a Japanese (only) speaking company. No matter how skilled the CEO, the language barrier will severely limit his effective control of the company. At the very least, there will be period of change for both the CEO and company. In a cell, though, only the company can change since the genome (CEO) only changes between generations and very slowly. The metabolism and genotype, then, coevolve together. Furthermore, the structure of the metabolism is dependent on the initial concentrations of chemicals in the cell. Both the Darwinian evolution of the genome and the Lamarckian evolution of the initial concentration set affect the metabolism. It appears that Lamarckian evolution of the initial concentrations is necessary for adaptation to difficult environments. Small problems can be solved in each generation taking the cell line closer to a complete solution of the environment. For example, a cell may produce chemicals that increase the rate of gene expression. Children of this cell benefit from the increased expression rate to produce other useful chemicals faster and in larger quantities.

Simulation of the Cell

Each cell simulation consists of the numerical integration of a large system of coupled nonlinear ordinary differential equations. There are variables in the system describing the concentration of each chemical in the system, as well as variables for the degree to which a gene is regulated and the number of cell wall molecules in the cell.

The system of equations may change over the life of a simulation if new chemical species are produced. This is an algorithmic device (“lazy evaluation”) used by Farmer, Kauffman, and Packard (1986) and Bagley (Bagley and Farmer 1991, Bagley, Farmer and Fontana 1991) to avoid calculating an infinite chemistry using finite computing resources. The initial chemical concentrations of the cell are the starting boundary conditions of the system of differential equations.

Cells exist from the interaction between a genome and metabolism (as directed from a set of initial chemical concentrations). The processes in the metabolism, too, derive from the initial concentrations and the genome. The cell differs from most phenotypes found in genetic algorithms literature (Holland 1992, Goldberg 1989) in that it is not static. The phenotype changes over the lifetime of the cell. Genes are regulated by the distribution of chemicals in the cell. Metabolic reactions that are specified by the genes currently expressed on the genome and genes that expressed slightly in the past modify this distribution of chemicals. Diffusion of chemicals through the membrane may also change concentrations of metabolic chemicals. The time varying nature of the phenotype arises from the numerical integration of the system of differential equations.

At the start of the simulation of a cell, the bases in the genome are parsed. The gene language is translated into terms in the system of differential equations (another language of sorts). Additional terms are added to the differential equations for the other metabolic processes, such as background diffusion and modification of the cell membrane.

The system of non-linear differential equations is numerically integrated using a Runge-Kutta algorithm with adaptive step size (Press et al. 1995). Integration occurs from $t = 0$, until t has passed a given value (the same for all simulations 1.5×10^5), or one of the variables in the phenotype has moved out of bounds. The latter event denotes cell death, an example of which would be a chemical species getting a concentration greater than 1.0×10^{-4} . After each time step, the variables are inspected to identify whether any chemicals have been created for the first time. That is, whether the concentration of a chemical has risen above the concentration equivalent to one molecule (Farmer, Kauffman, and Packard 1986). If this has occurred, terms are added to the differential equations where this new chemical interacts with the rest of the cell. As well, we determine whether the number of water molecules in the cell has changed markedly. This can happen as a result of a change in the number of cell wall molecules surrounding the cell. Number of cell wall molecules is a primary indicator of the surface area of the cell membrane and hence, the size of the cell. Cell size is the primary attribute that a cell must control in our simulation.

If the number of water molecules has changed, the variables representing concentrations of chemicals in the cell are adjusted accordingly. As we shall see, this is required because the concentration of a chemical is defined as the ratio of the number of molecules of the chemical in the cell to the number of water molecules in the cell. The refinement of a concentration as the result of a change in number of water molecules in the cell is usually so small as to no effect on the simulation.

The model as a coevolutionary system

An interesting philosophical question to ask relates to the evolution of each part of the cell model. Above, we stated that the genome follows Darwinian evolution but that the initial metabolic conditions evolved according to a Lamarckian scheme. We can discuss the evolution of each part of the cell model as happening separately. Can we then describe the cell model as the coevolution of the genome and initial metabolic conditions?

Solomon et al. (1993) define coevolution as “the interdependent evolution of two or more species that occurs as a result of their interactions over a long period of time”. This biological definition does not sit well with our aggregated simulation. Kauffman (1993) provides broader guidelines. He says that the difference between coevolving systems and evolving systems is that, in simple evolving systems, individual components do not replicate. This means that selection may act only on the system in its entirety. In coevolving systems, components replicate. This allows selection to act on parts of the system as well as on the whole. Furthermore, he says,

“there is a fundamental difference between simple adaptive evolution and coevolution. Evolution on a fixed fitness landscape ... is similar to the behaviour of a physical system on a well-defined potential energy landscape. In both cases, the attractors of the ‘adaptive’ process are local optima which are single points. In a coevolutionary process, however, the adaptive landscape of one actor heaves and deforms as the other actors make their own adaptive moves. Thus coevolving behaviour is in no way limited to attaining point attractors which are local optima, nor is it even clear that coevolving systems must be optimizing anything whatsoever.”

Many examples of coevolution can be found in Nature. In mutualism (or symbiosis), both species involved benefit from their relationship. An example of mutualism is the pollination of flowering plants by insects (Solomon et al. 1993): the plant achieves pollination whilst the insect is nourished. Kauffman (1993) gives another: multicellular organisms consist of coevolved cells. In other forms of coevolution, one organism gains at the expense of the other. The standard example of this form of relationship is that of lions and antelopes or, equally, cheetahs and gazelles (Dawkins 1986). As well as ‘arms races’, such as these, host-parasite relationships can arise. Some examples are not completely clear. Lichen (Solomon et al. 1993, Dawkins 1989), the symbiotic association between a phototroph (an organism that photosynthesises) and a fungus, used to be the definitive example of mutualism: neither organism could exist separately in their environment, but together they survive in many habitats. However, their relationship may be one of controlled parasitism of the phototroph by the fungus (Solomon, et al. 1993).

Our cell model certainly doesn’t conform to the standard examples of coevolution in ecosystems. An example from Alberts, Bray, et al. (1994) concerning the origin of eukaryote cells, though, is interesting. The endosymbiont hypothesis states that

mitochondria and chloroplasts evolved from bacteria over a billion years ago. It says that eukaryotic cells began as anaerobic organisms without mitochondria or chloroplasts. At some stage (possibly when oxygen entered the atmosphere), these organisms established a stable endosymbiotic relationship with bacterium whose oxidative phosphorylation system enabled them survive in oxygen rich environments. These bacteria were absorbed into the proto-eukaryote cells to become the predecessors to mitochondria. Mitochondria organelles contain their own genetic information, although some has migrated to the eukaryote nucleus over time. (Alberts, Bray, et al. 1994)

Applying our model to Kauffman's definition of coevolution, the genome and metabolism in our cell model replicate using separate mechanisms. Selection, however, does not truly occur on each separate part. The fitness function we use scores a cell based on a combination of features from both parts. Fitness of one component of the cell affects fitness of the other: unfavourable initial metabolic conditions may cause an otherwise fit genome to perform badly. On balance, we feel that our cell model can be classified as a coevolving system of genome and initial metabolic conditions.

3.2 Chemistry Model

All chemical reactions occur within an abstracted model of chemistry that is the same for all cells in the simulations. The chemical model we employ is similar to that used by Farmer, Bagley and others (Farmer, Kauffman, and Packard 1986, Bagley and Farmer 1991, Bagley, Farmer and Fontana 1991). Bagley and Farmer (1991) describe two types of artificial chemistries: one completely random and the other wholly deterministic. In the stochastic chemistry, the choice of whether a reaction is catalysed by a particular chemical is random. The structure of the catalysing chemical is arbitrary. This differs from real biology, where the conformation of an enzyme governs the reaction(s) it can catalyse. The deterministic chemistry takes the notion of shape to the extreme. In that model, the shape of the enzyme determines what reactions can be catalysed and their speed. Bagley believes that real chemistry lies between these two extremes.

The artificial chemistry we use builds on Bagley's deterministic model. However, we do not allow every kind of chemical to act as a catalyst. We class chemicals into three types (ordinary chemicals, enzymes and carrier molecules) and allow only enzymes to catalyse reactions. There are other kinds of reactions and matching algorithms in our artificial chemistry as well. These model other processes in the cell. It is important for the chemistry to be the same for all cell simulations in the genetic algorithm. No evolution can occur if the underlying reactions are different between parent and child.

3.3 Chemical Model

Chemicals are the basic elements of a cellular metabolism in the simulation. They are an abstraction of polypeptide molecules found in nature. We base our chemicals on the model proposed by Farmer, Kauffman, and Packard (1986) and used by Bagley and Farmer (1991) and Bagley, Farmer and Fontana (1991) but extend it to allow a simple type system as shown in Table 3.1.

Typing chemicals allows us to simplify the model. For example, any chemical produced from gene expression has type 'enzyme'. Its properties are fixed to those of enzymes. Our chemical model is more abstract than Farmer's and so, more removed from Nature. Although we realise that there is more to an enzyme than a simple list of the monomers comprising it, this is a first identifier of its properties. This simple abstraction can transfer the real complexity of an enzyme's shape into a type and just store the shape information that determines its precise properties. Each chemical, then, consists of a type, a shape and a concentration.

Table 3.1: Types of chemicals

Types of Chemicals	Symbol
ordinary chemicals	o
enzymes	e
carriers (in the cell)	ic
Subsidiary types	
bound ordinary chemicals (ie. in a reaction)	_o
bound enzymes (ie. in a reaction)	_e
partially built enzymes and carriers	p
carriers (out)	xc
bound carriers (in) - partly diffused	ib
bound carriers (out) - partly diffused	xb
Other variables (not chemicals)	
Operon switches	g
Number of water molecules	w
Number of membrane molecules	m

Chemical Type

The type of a chemical specifies how it interacts within the cell. For example, an enzyme can catalyse reactions, but not otherwise take part in a reaction. Similarly, carriers can act as transmembrane gates for ordinary chemicals. The metabolic processes we have modelled and their relationship to the chemical types is given in Table 3.2.

Metabolic processes	Can be performed by
Taking part in reactions	ordinary chemicals only
Catalysing reactions	enzymes only
Expressing genes	ordinary chemicals and enzymes

Table 3.2: Properties of chemicals**Chemical Shape**

The ‘shape’ of a chemical consists of a list of digits. Each digit represents a monomer (amino acid) making up the polymer (protein). Taken together in order, the digits determine the properties of the chemical. When catalysing a reaction, the closer the lists of digits comprising the reactants match the enzyme’s list, the faster the reaction can occur (section 4.2). (Catalysis doesn’t work precisely like this, but our abstract model will use this level of detail for ‘matching’). Gene regulation works similarly using matching. The closer the shape of the chemical is to the promoter region of the operon, the more effectively that chemical can regulate the operon. The matching algorithm used to quantify this resemblance between two strings is based on a scheme described by Farmer, Packard and Perelson (1986).

We use a different strategy for calculating expression of genes and when constructing and destructing the membrane. Here, we choose arbitrary chemical shapes that perform each task perfectly. The closer the resemblance between a given chemical and this Platonic shape, the more efficiently the chemical performs the task. This allows a close family of chemical species to act in a similar way and aids hill climbing.

Matching schemes are used because determination of the properties of a chemical is a difficult task. Rather than introduce a model of polarity and molecular bonding, a simple first approximation is simply to compare strings of digits.

Chemical Concentration

Each chemical species in a cell has a real valued concentration in the interval $[0,1]$ denoting the relative amount of that species in the cell. Any chemical with a concentration above 1.0×10^{-4} is assumed to be lethal to the cell. The lethal concentration

threshold is arbitrarily chosen but is plausible. Its actual value is not as important as its relationship to other parameters.

The definition of concentration most often used in chemistry is that of a mass density (or a volume density). Mass density is an unsuitable definition for our model because we don't directly model the masses of the monomers and water molecules. Instead, we use a number density. The concentration of a chemical is the ratio of the number of molecules of the chemical to the number of water molecules in the cell. This ratio may well be an approximation to the mass density, although our concentrations tend to be an underestimate for larger molecules because we assume the volumes (or masses) of molecules are the same over all species.

We use a simple notation for writing chemical species. $xnnn$ describes a chemical species of type x , with shape nnn , where the given values for x are listed in Table 3.1.

Chapter 4

MODEL OF METABOLISM

Mr. Verloc ... opened the door ... and thus disclosed the innocent Stevie ... drawing circles, circles, circles; innumerable circles, concentric, eccentric; a coruscating whirl of circles that by their tangled multitude of repeated curves, uniformity of form, and confusion of intersecting lines suggested a rendering of cosmic chaos, the symbolism of a mad art attempting the inconceivable.
- Joseph Conrad, "The Secret Agent"

In this chapter, we discuss the model of the cellular metabolism in broad terms, and describe two aspects of it, metabolic reactions and protein degradation, in more detail.

The previous chapter provided an overview of the cell simulation but deferred descriptions of each component. Two elements of the metabolism of a cell are specified here. Account of gene expression and regulation is reserved to chapter 5, after the genome model has been discussed. Descriptions of the diffusion of chemicals through the cell membrane and cell growth are deferred to chapter 7.

Initially, a description of the model of the cell metabolism is given and the metabolic processes accessible to the cell listed. Metabolic reactions in the cell are described and an encoding of the reactions into a system of differential equations, after an approach by Farmer and Bagley (Farmer, Kauffman, and Packard 1986, Bagley and Farmer 1991, Bagley, Farmer and Fontana 1991), is detailed. A model of protein degradation based on biology is introduced and an encoding into differential equations designated.

4.1 Brief Description of Cell Metabolism

The cell metabolism consists of a set of real valued variables denoting the state of the cell and an equal number of non-linear ordinary differential equations (chemical kinetics equations) providing relationships between elements of the cell. Most variables represent the concentration of chemicals in the cell. Other variables specify the expression capability of an operon, or the number of cell membrane or water molecules in the cell.

All models of processes occurring in the metabolism are encoded as terms in the system of differential equations.

As we stated in chapter 3, five distinct categories of processes in the metabolism are modelled: metabolic reactions, gene expression and regulation, protein degradation, cell growth and diffusion of chemicals through the cell membrane.

Metabolic reactions are concerned with the production of new chemicals (ordinary chemicals) from the condensation of two existing chemicals or the cleavage of a chemical. All reactions in the cell are enzyme catalysed.

Gene expression is the process by which new proteins (enzymes or carriers) are introduced into the cell. The speed of protein production can be controlled by the presence and concentration of particular chemicals in the cell.

Protein degradation is concerned with removing proteins from the cellular metabolism. Proteins, in this model, have a half-life. (Alberts, Bray et al. 1994) When this time has expired, the protein breaks down into its constituent parts.

Cell growth involves the addition of molecules to the cell membrane. The size of the cell membrane can also be decreased. Growth or contraction of the membrane can be managed by the presence of particular chemicals in the cell.

Diffusion of chemicals through the cell membrane occurs in two ways: slow background diffusion and a much faster traversal through transmembrane gates.

4.2 Metabolic Reactions

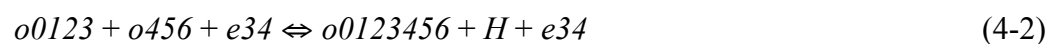
Some molecules in the cell are more ‘useful’ than others. This depends on their shape. Apart from diffusion and production of custom-built proteins from gene templates, the only way for a cell to obtain a particular molecule is to fabricate it using other molecules to which it has ready access. Both diffusion and gene expression offer limited access to novel chemical species. As regards to diffusion, a cell is at the mercy of its environment and, in our model, usually only small ordinary molecules diffuse through the membrane. As for gene expression, only proteins may be constructed in this fashion. The most reliable method for our cells to fabricate particular families of ordinary chemicals is by metabolic reactions.

The chemical reactions in this model are an abstraction of polymer condensation and cleavage reactions from biology. Each reaction models an equilibrium of the joining of two polymers under catalytic pressure from an enzyme.



where A and B are substrate molecules, C is a product, E an enzyme and H water.

For example, the reaction



describes adding the polymer $o456$ to the tail of the polymer $o0123$ with catalysis from the enzyme $e34$ (condensation). Note that the first part of the enzyme (3) must match the tail of the first reactant and the end of the enzyme (4) must match the head of the second reactant. The reaction can equally be viewed as a cleavage of the long polymer into two smaller fragments. The relative concentrations of the reactants and products determine the direction the reaction proceeds. Enzymes that closely match the reactants

and product allow the reaction to take place faster than enzymes that don't match as well. Rates associated with enzyme matches are derived and discussed later.

We use a similar matching chemistry to Farmer, Kauffman, and Packard (1986) and Bagley et al. (Bagley and Farmer 1991, Bagley, Farmer and Fontana 1991) except that we do not employ complementary matching. Noncomplementary matching is easier for humans to read than complementary matching and produces similar results. Nature, of course, uses complementary matching, but this is a result of the physics and chemistry involved.

Like Farmer, Kauffman, and Packard (1986), we model only enzyme catalysed reactions. This differs from Bagley and Farmer (1991). We assume no spontaneous reactions occur. This assumption is validated because spontaneous reactions occur at a vastly slower rate than enzyme catalysed reactions. The enzyme acts to quicken the reaction. The more specific the enzyme to the reaction, the faster the speed up that occurs. Specificity is dependant on the degree of matching between the enzyme and the substrates or product.

Adding New Reactions

All reactions in a cell can be combined into a directed graph to show the possible biochemical pathways allowable in that cell. Chemicals comprise the nodes of the graph and reactions are the edges. The actual pathways taken through this graph of possibilities depend on which genes express and what chemicals are in the cell or diffuse through the membrane. In a similar way to Farmer's and Bagley's models, the system changes over time and new chemicals may be produced in detectable concentrations (ie, at least one molecule of it exists in the cell). This would represent the mean field estimation that one (or more) molecules were present. When this happens, additional reactions are added to the reaction graph to deal with the new reactions that are now possible. It is not possible to calculate all the reactions that are possible from the outset because the number grows large very quickly. We have followed the method used by Farmer, Kauffman, and Packard (1986) where growth of the reaction graph (possible reactions) is tempered by the chemicals actually existing in the system in

quantities of at least one molecule. This lazy scheme is a good method of handling an infinite chemistry in a finite way.

Farmer's situation, where any chemical can act as a catalyst, is different to ours. No new enzymes are ever added to our cells. Enzymes are trapped in the cell because we deem them too large to diffuse through the membrane. A mother cell, though, bequeaths her enzymes to her children. Because no enzymes are added after the birth of the cell, our chemistry is more constrained than Farmer's is. Old enzymes tend to be reinterpreted and used in different ways over the lineage of a germ line. The reaction graph, though, still expands with a computational explosion because more complicated products appear that are still catalysed by the enzymes. This computational explosion is eventually controlled using the genetic algorithm. We select only for cells with fairly simple reaction graphs.

Calculation of Rates for Enzyme Catalysed Reactions

We calculate the speed up of a reaction due to the presence of a catalysing enzyme in the same way as Farmer, Kauffman, and Packard (1986). We match the enzyme against the substrates as shown in figure 4.1.

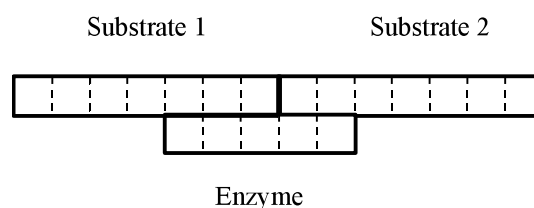


Figure 4.1: Matching an enzyme against two substrates.

Because there must be at least one base in the enzyme matching each substrate, the enzyme can match against the substrates in $l-1$ positions, where l is the length of the enzyme.

For each position that the enzyme can match against the substrates, we determine the number of matches between the enzyme and substrates. We determine the specificity of the enzyme if the enzyme binds at least once to each substrate.

The enzyme speed up v is determined using

$$v = \sum_{i \in I} \frac{C_s}{P(X \geq m_i)} \quad (4-3)$$

where C_s is the specificity coefficient (110),

I is the set of possible matching positions,

m_i is the number of matches between the enzyme and substrates at position i , and

$P(X \geq m_i)$ is the probability that m_i matches or more could arise randomly (a measure of enzyme specificity).

$P(X \geq m)$ is given by the binomial distribution

$$P(X \geq m) = \sum_{s=m}^n p^s q^{n-s} \frac{n!}{s!(n-s)!} \quad (4-4)$$

where n is the length of the enzyme,

$p = 0.1$ is the probability of monomers matching, and

$q = 1-p$ is the probability of the monomer not matching.

Differential Equations for Metabolic Reactions

Before writing differential equations to model reaction (4-1), the problem of saturation must be solved. We use a solution by Farmer, Kauffman, and Packard (1986). Reaction (4-1) is split into four irreversible reactions



\overline{CE} is the bound product-enzyme complex. \overline{ABE} is the bound substrate-enzyme complex. k_f is the rate constant determining the speed of the forward reaction without enzyme catalysis. That is, condensing the two substrate polymers into a longer product. k_r is the rate constant for the cleavage reaction without enzyme catalysis. Finally, k_u is the rate constant determining the speed that the enzyme-substrate (or enzyme-product) complex will disassociate into the separate enzyme and substrate molecules. Typical values are given in table 4.1. These values are different to those given in Farmer, Kauffman, and Packard (1986), but their relative values are similar. These parameters are proportional to the time scale. The actual value of the parameters is not as important as their relative values. Farmer, Kauffman, and Packard (1986) selected values that were consistent with Nature.

Table 4.1: typical values for rate constants

k_f	1.0×10^{-5}
k_r	1.0×10^{-4}
k_u	1.0×10^{-2}
ν	1000-110000

With many species and enzymes, the number of intermediate species such as \overline{CE} and \overline{ABE} can become very large. The simulation would quickly become intractable. Farmer, Kauffman, and Packard (1986) addressed this problem by making the simplifying assumption that k_u is the same for all bound complexes. With this assumption, it becomes possible to reduce the total number of variables needed to represent the metabolic reactions to twice the number of chemical species participating

in the reactions. One set of variables represents the concentrations of the chemicals. The other set represents the contribution for each of the bound forms including a chemical.

Each chemical species, then, yields two differential equations and variables: its concentration (x_i) and the combined concentration of all the complexes into which it is bound (\bar{x}_i).

The differential equation governing the concentration of the chemical species is (after Farmer, Kauffman, and Packard (1986)):

$$\frac{dx_i}{dt} = k_u \bar{x}_i + \sum (\text{reaction terms}). \quad (4-7)$$

where (reaction terms) is a sum of terms for reactions x_i participates in.

- $k_f v ABE$ is added in when x_i acts as a substrate (A or B) in a reaction.
- $k_r v CHE$ is added when x_i acts as a product (C).
- $vE(k_f AB + k_r CH)$ is added when x_i is an enzyme (E) in the reaction.

Note that this last term corrects a sign error introduced into Bagley and Farmer (1991). This error was not present in Farmer, Kauffman, and Packard (1986).

The differential equation for the sum of bound complexes containing species i is:

$$\frac{d\bar{x}_i}{dt} = -k_u \bar{x}_i + \sum (\text{reaction terms}) \quad (4-8)$$

- + $k_r v CHE$ is added x_i is a substrate (A or B) in the reaction.
- + $k_f v ABE$ when x_i is a product (C)
- + $vE(k_f AB + k_r CH)$ when x_i is an enzyme (E).

4.3 Protein Degradation

In our model, enzymes and carriers cannot participate in reactions as reactants. The metabolic reaction graph, therefore, cannot be used to remove them from the cell. They soon reach high concentrations and poison the cell. Biological cells do not have reactions that are as structured and constrained as ours. We, therefore, looked towards biology to determine the kinds of metabolic processes to add to the cell for protein breakdown.

Researchers have identified a number of mechanisms for protein breakdown in biological cells (Alberts, Bray et al. 1994) such as ubiquitin-dependent proteolytic pathways, proteosomes and enzymes that alter a protein's N-terminus (Alberts, Bray et al. 1994). With the aim of keeping the model relatively simple, we associate a half-life with enzymes and carriers. This is modelled by an irreversible first-order reaction that breaks each protein into its constituent monomers. The same rate constant (typically 0.1) is used for all reactions.



This reaction example results in four additional terms to the system of differential equations.

$$\frac{d[e3123]}{dt} = \dots - k[e3123] \quad (4-10)$$

$$\frac{d[o1]}{dt} = \dots + k[e3123] \quad (4-11)$$

$$\frac{d[o2]}{dt} = \dots + k[e3123] \quad (4-12)$$

$$\frac{d[o3]}{dt} = \dots + 2k[e3123] \quad (4-13)$$

where k is the protein degradation rate constant mentioned above and $[n]$ is the concentration of chemical species n .

Chapter 5

MODEL OF GENOME

Computationally, the DNA does not only constitute a reservoir mechanism for heredity and for the self-construction of the system, but a problem-solving device continuously used: it constitutes the main source of adaptation to the demands posed by the internal and external environment.

- Pedro C. Marijuán, (1994)

In this chapter, we describe the structure of the cell's genome, the manner in which it encodes information and the genetic operators that can be applied to the genome to change the information it contains.

Chapter 3 examined the relationship between the two main components of the cell model: the metabolism and the genome. Detailed examination of the genome was deferred to this chapter. Chapter 6 comprehensively describes the relationship of the genome to the metabolism.

Initially, in this chapter, we discuss the particular structure of the genome and its relationship to Nature and genetic algorithms. Next, we describe how genomes encode sentences in a parallel gene language. Finally, we examine the genetic operators that may be applied to the genome. In particular we introduce a new biologically inspired algorithm for the inversion genetic operator.

5.1 Genotype

Our model of the cell's genome has been inspired from biology (Solomon et al. 1993, Alberts, Bray et al. 1994, Gardner, Simmons and Snustad 1991). We extend the

standard genome model commonly used in genetic algorithms (Holland 1992, Goldberg 1989) with more abstractions from biology.

It is well known that in Nature DNA molecules encode protein sequences that a cell needs to function (Solomon et al. 1993, Alberts, Bray et al. 1994, Gardner, Simmons and Snustad 1991). A DNA molecule consists of two strands bound together into a double helix. Each strand has a backbone of sugar molecules connected together by phosphate molecules. One of four bases (adenine, thymine, guanine or cytosine) binds to each sugar molecule. The order of the bases along a strand is arbitrary. Bases form complementary pairs: a molecule of adenine can bind weakly to a molecule of thymine, and similarly guanine can bind to cytosine. The bases of one strand are thus complementary to the bases in the other strand. The complementary matching between opposite bases causes the two strands to bind together into one DNA molecule. Apart from occasional errors, one strand completely determines the bases appearing in the other strand. A strand is normally read three consecutive bases (a codon) at a time. Each codon is converted into one of twenty amino acids that make up cellular proteins. The sequence of codons specifies a protein molecule that the cell can use for catalysis of reactions, for a structural function, or for some other specialised use.

As in Nature, our genomes contain two strands. This feature admits a small increase in data compression, but more importantly allows a simple, biologically inspired algorithm for the inversion genetic operator (described in section 5.7). The main benefits that biological genomes obtain from having two strands are that the DNA molecule is structurally more stable, and replication of DNA is made conceptually simpler and more error resistant. Use of simpler structures when more complex ones are not necessary is superior because they tend to use less energy, are easier to build and there are less ways for them to be broken. Neither simplicity or error resistance is important for artificial genomes because they exist in an error free simulation. This is probably the reason why double stranded genomes do not appear in the literature of evolutionary computation.

The double stranded nature of genomes is different to diploidy (Holland 1992, Gardner, Simmons and Snustad 1991), where each organism has two sets of chromosomes. In Nature, a diploid organism has two copies of a chromosome, each one consisting of two

strands. Alleles are variations of a gene found between the two chromosomes, *not* across the two strands.

Three genetic operators are applied to the genome: recombination, mutation and inversion (Holland 1992, Goldberg 1989).

Another extension to the traditional GA genome is our introduction of a parallel gene language. This language specifies the molecules that the cell may produce using the genome as a template, as well as the metabolic chemicals that may be used to regulate gene expression. ie, transcription is a mapping from codons to monomer types.

The gene language is an abstraction of the operon model of gene regulation described in 1961 by Jacob and Monod (Gardner, Simmons and Snustad 1991, Alberts, Bray et al. 1994). We view the genome as consisting of operons separated by non-coding regions. An operon is the smallest regulatory unit of the genome. This is a region of the genome comprising a promoter region, which stores regulatory information, followed contiguously by one or more genes. Each gene in the operon is regulated in the same way. We follow the “one gene-one enzyme” concept of molecular genetics (Gardner, Simmons and Snustad 1991), which states that each gene codes for only one enzyme. Each gene in our model, therefore, encodes one enzyme. Our model allows selective facilitated diffusion of molecules through the cell membrane using special transmembrane molecules that are also encoded in genes. We view these as specialised enzymes. This means that the “one gene-one enzyme” concept is still valid. In Nature, once it was discovered that enzymes could be built from smaller proteins, each of which could be encoded in separate genes, the “one gene-one enzyme” tenet was modified to “one gene-one polypeptide”. We ignore this modification since our model does not allow enzyme construction from smaller polypeptides.

5.2 Double Strand Bit Encoding

In the model, genomes comprise a simple bit string for their underlying Boolean representation. Imposed above this are two logical strands, as in Nature’s DNA molecule. (See figure 5.1)

The double strand encoding scheme we use abstracts the encoding scheme from biology where each strand of a DNA molecule can only be read in only one direction and where one strand is the complement of the other. We use a simple bit string to represent the first strand. It is read from bit position 0 to n . The second strand is the bit complement (ones complement) of the first string and is read in the opposite direction (n to 0). As the second strand is completely determined from the first, only one strand needs to be stored. The total number of bits in the genome, therefore, is twice the number of bits in one strand. It is permitted for a segment of the genome to code parts of two different genes: once forwards on the first strand, then backward complementarily on the second strand. We keep the number of bits in a strand divisible by four so that bits can be grouped into nibbles (four contiguous bits equal one nibble). Typically, our genome has 400 bits in each strand.

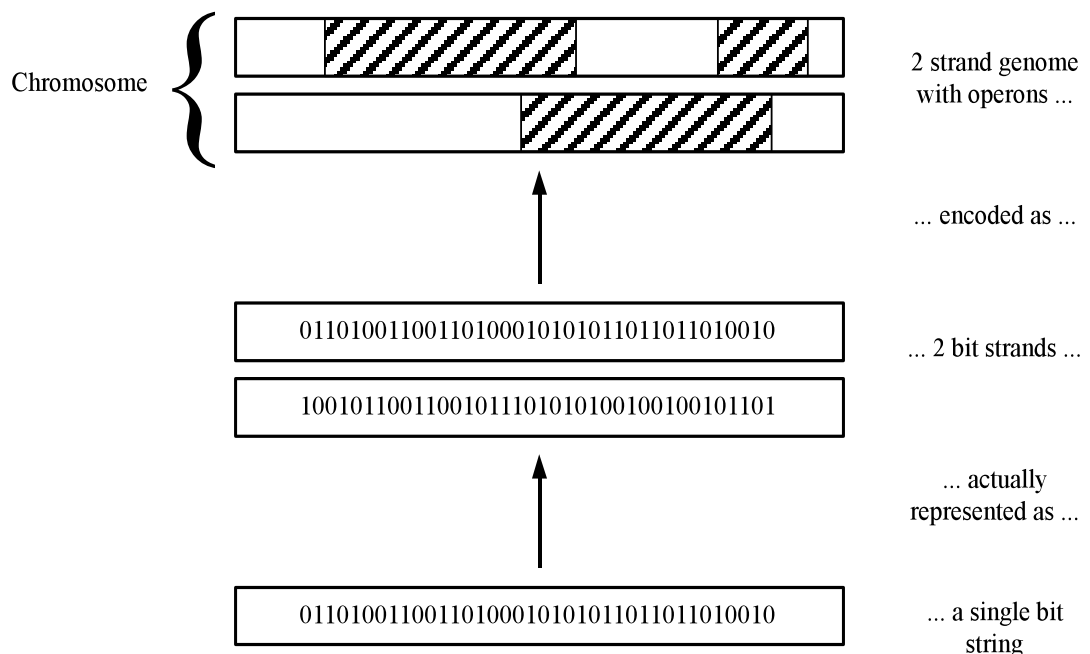


Figure 5.1 Representation of genome

One genome is used in the cell. This equates to one chromosome. Our model employs a haploid genome, although it is extensible to diploidy. By this, we mean that the cell

holds one copy of the one chromosome (like prokaryotes) rather than the two copies that are found in eukaryote cells.

5.3 Genome Bit Encoding

Dividing the bit strings into four bit blocks allows sixteen coded words – called bases. Nature employs the equivalent of two bit blocks to have four bases and then, often, reads three blocks at a time (a codon) to allow sixty-four symbols (see figure 5.2). We read bases one at a time to give sixteen symbols. Sixteen bases were chosen for our model because they can conveniently be divided into ten bases to match the number of monomers in our chemistry and six bases for use in the gene language described below.

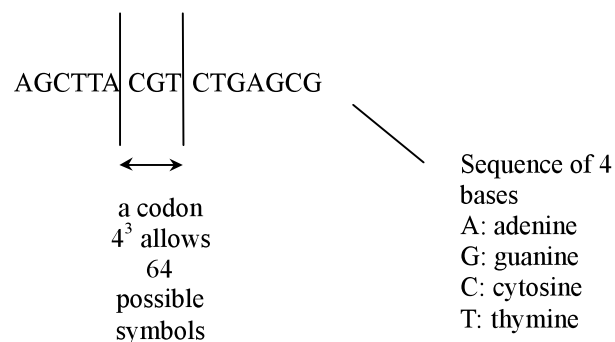


Figure 5.2 Encoding scheme from Nature

Gray Coding

Researchers in evolutionary computation (Back 1996, Goldberg 1989) typically code integers into bit strings on the genome using Gray code. This encoding scheme ensures that a one bit mutation to the genome changes the coded integer by only 1. Integers coded onto the genome using their ordinary binary value can change by much larger values as the result of a one bit mutation. More alarmingly, the amount the number changes by is different depending on the actual bit changed. The lowest order bit changes the encoded number by one; the second lowest order bit changes it by two, and so on.

We do not Gray code nibbles. The motivation behind this choice is that there is no natural order of monomers and other gene language words. There is no ordering relationship between different bases. The bases represent discrete symbols.

Base Coding

The nibbles coding [0 = 0000, 9 = 1001] are used to specify monomers to build into molecules or as signatures used for gene regulation (see table 5.1).

The remaining six bases code for four special codes: *<start operon>* (nibbles [a = 1010, b = 1011]), *<start enzyme>* (nibbles [c = 1100, d = 1101]), *<start carrier>* (nibble e = 1110) and *<end operon>* (nibble f = 1111).

The *<start operon>* symbol marks the place on the genome where an operon begins. Similarly, *<start enzyme>* signifies that the bases following will code for an enzyme molecule. The *<start carrier>* symbol specifies that bases following will code for a carrier molecule. The *<end operon>* symbol marks the end of an operon. There are other ways to mark the end of an operon as well.

The reader will notice that more nibbles are available for the *<start operon>* and *<start enzyme>* symbols than the other symbols. Although there is redundancy in Nature's genetic code, there is no biological reason for our redundancy. It appears likely to be better for the genetic algorithm to avoid codes that don't decode to symbols: the genome search space is reduced to exclude invalid genomes (or at least invalid genes).

Table 5.1 Base codings

Nibble Value	Symbol	Meaning
0000 (0)	base 0	Monomer value or promoter signal code
0001 (1)	base 1	
0010 (2)	base 2	
0011 (3)	base 3	
0100 (4)	base 4	
0101 (5)	base 5	
0110 (6)	base 6	
0111 (7)	base 7	
1000 (8)	base 8	
1001 (9)	base 9	
1010 (a)	<start operon>	Signals the start of an operon
1011 (b)	<start operon>	
1100 (c)	<start enzyme>	Signals the start of gene coding for an enzyme. That is, a gene that will build a chemical that can catalyse reactions.
1101 (d)	<start enzyme>	
1110 (e)	<start carrier>	Signals the start of a gene coding for a carrier. That is, a gene that will build a chemical that allows facilitated diffusion of chemicals through the cell membrane.
1111 (f)	<end operon>	Signals the end of an operon.

There is no bias against the <end operon> symbol because there is another way to announce the end of an operon. A <start operon> symbol found inside an operon ends that operon and starts another. Similarly, the end of the genome automatically ends the current operon. This still leaves more encodings for carriers. However, this bias appears only towards randomly generated nibbles rather than chromosomes with an evolved ancestry. In this system, random nibbles only occur when a cell in the initial population is randomly generated and when genomes are mutated. In both cases, genomes that

don't code well are soon eliminated from the population. So the bias is only apparent rather than actual after some selection pressure has applied over time.

Goldberg's Coding Principles

Goldberg (1989) specifies two coding principles or heuristics to follow to ensure a good genome encoding scheme.

The first heuristic is the *principle of meaningful building blocks*. In Goldberg's words, "the user should select a coding so that short, low-order schemata are relevant to the underlying problem and relatively unrelated to schemata over other fixed positions". Schemata in our encoding scheme are built from the nibble codes. Meaningful building blocks will result because a language ties the codes together. The language relates schemata over other fixed positions, though. This is difficult to avoid. For example, a *<start operon>* code forms an association with all following codes. As our model follows from biology, this is true of Nature too.

Goldberg's second principle is that of minimal alphabets. Here, "the user should select the smallest alphabet that permits a natural expression of the problem". This is true of our encoding scheme.

5.4 Biological Motivations

The expression of genes into proteins can be divided into two main tasks: transcription of genes and translation into proteins (Gardner, Simmons and Snustad 1991, Alberts, Bray et al. 1994). Transcription of genes is the process of copying the information stored in the bases of a DNA molecule onto a shorter and more mobile mRNA molecule. This typically occurs in the nucleus of eukaryote cells (Alberts, Bray et al. 1994). Translation is the process of converting the information stored in a mRNA molecule into the amino acid sequence of a protein molecule. Translation typically occurs outside the cell nucleus in the endoplasmic reticulum (ER) (Alberts, Bray et al.

1994). Both processes are extremely complicated and researchers do not understand either fully.

Transcription

During transcription, genes from the DNA are transcribed onto mRNA (messenger RNA) molecules. mRNA is a similar molecule to DNA, in that bases can be arranged arbitrarily along a sugar and phosphate backbone. However, it does not have a double stranded structure and the base uracil is used in mRNA instead of thymine.

Complementary matching ensures that the mRNA copy is a ‘negative’ of the bases in the DNA strand.

The nature of the DNA molecule forces transcription of a strand to proceed in one direction only. As well, the two strands are antiparallel. That is, they have opposite chemical polarity. Because of these constraints, each strand is read in the opposite direction than its matching strand, much like a two lane road. Genes are transcribed from either strand, but usually only the one strand for an entire operon. This strand is called the template strand. A complicated system of molecular (enzymatic) machinery finds the start of an operon and unwinds the double helix. It separates the two strands and transcribes bases of the template strand, building a mRNA molecule. As the machinery moves along the DNA molecule, it rewinds the double strands together again behind it. Transcription starts downstream from the transcription initiation site in the promoter region of the gene and continues to a termination sequence.

Translation

Once the mRNA molecule is complete, it moves out of the cell nucleus to the ER and translation begins. Codons of the mRNA are translated into amino acids, which are linked onto the end of a growing protein molecule.

Translation of a codon into an amino acid is accomplished with the help of tRNA (transfer RNA) molecules. These molecules associate a specific amino acid with the

matching codon(s). Each tRNA molecule has a three base anticodon that binds to the mRNA codon using complementary matching. The actual linking of amino acids into a protein molecule is effected by a complex group of molecular machinery (proteins and enzymes) called a ribosome. A ribosome walks along the mRNA molecule and allows tRNA molecules to match an amino acid to each codon. It then binds the amino acid onto the end of a growing protein molecule. Further modification of the protein chain, transport within the cell, and priming are required before the enzyme is ready for use.

Gene Regulation

The process of gene expression is complicated because the genome can be regulated in many ways and places (Alberts, Bray et al. 1994). The primary kind of gene regulation we built into our model is transcriptional control.

Transcriptional control regulates the expression process at the point when the DNA is transcribed into a mRNA molecule. Transcription of the gene is controlled by a sequence of DNA near the transcription initiation site. Methods of control, however, are different between prokaryote and eukaryote cells. In prokaryote cells, the regulatory region acts predominantly as a switch, turning transcription of the genes on or off and acting on only a few signals. In eukaryote cells, this region is complex and may be influenced by other regions of DNA many thousands of bases away from the gene, acting on many kinds of signals. In this model, we are interested only in the fact that a gene may be switched on or off. Our model is not large, so one gene doesn't need to act on many signals. With these simplifying assumptions, we chose to implement a model of transcriptional control believed to be used in prokaryotic cells. As stated above, this is the operon model proposed by Jacob and Monod.

Regulation occurs using gene regulatory proteins, which are able to read the bases in the DNA molecule from the outside and bind strongly and very specifically to the part of the DNA molecule they match.

An operon is a contiguous region of DNA that consists of a promoter region followed by one or more genes. All the genes in an operon are regulated in the same way.

Overlapping the promoter sequence is a region of DNA called the operator, to which gene regulatory proteins bind. For transcription to commence, the transcription machinery needs to bind to a transcription initiation site in the promoter region. Gene regulatory proteins that are bound to the operator may inhibit or activate transcription by physically interacting with the transcription machinery's bonds to the transcription initiation site.

Inhibition of the gene occurs when a gene regulatory protein binds to the operator region of an operon so that the enzyme transcription machinery can no longer bind and start transcription. Activation of a gene, on the other hand, occurs when the promoter region of a gene is only marginally functional on its own. A gene regulatory protein bound near the promoter can help the transcription machinery to attach to the DNA. Genes can have both an inhibitor and an activator to allow control from two signals although we have not modelled this situation.

Some genes, which produce enzymes that are in continual demand, are always active. These are called constitutive genes. We also model this kind of gene.

Two other forms of gene regulation are modelled. These are regulation of the amount of mRNA (or its analog in our model: 'spiders') available to the cell, and competition in the metabolism for the enzyme building blocks (amino acids in cells, the monomers in [0,9] for our model).

This concludes our brief digression into the natural system we model.

5.5 The Gene Language

In our model, a simple gene language imposes semantics on the raw genome. The genome consists of operons interspersed with semantically meaningless data. This meaningless data is significant however, because it stores potential new genes and parts of old operons that are no longer used. These non-coding regions produce genes when a fortunate mutation, inversion or crossover occurs. Ruined operons store a garbled

history of the genome over past generations and may allow the cell to rediscover past functionality if required.

Operons are the smallest segment of a genome able to be regulated in our model. As we stated before, they comprise a promoter region followed by one or more genes. In the model, the promoter region is a sequence of bases that mark the start of the gene and allows gene expression and regulation to occur. Biological promoter sequences in bacteria consist of around 40 bases, but the sequence may be different among prokaryote and eukaryote cells and even between different operons (Gardner, Simmons and Snustad 1991). A schematic for operons is illustrated in figure 5.3a. Figure 5.3b shows the bases comprising an example of an operon.

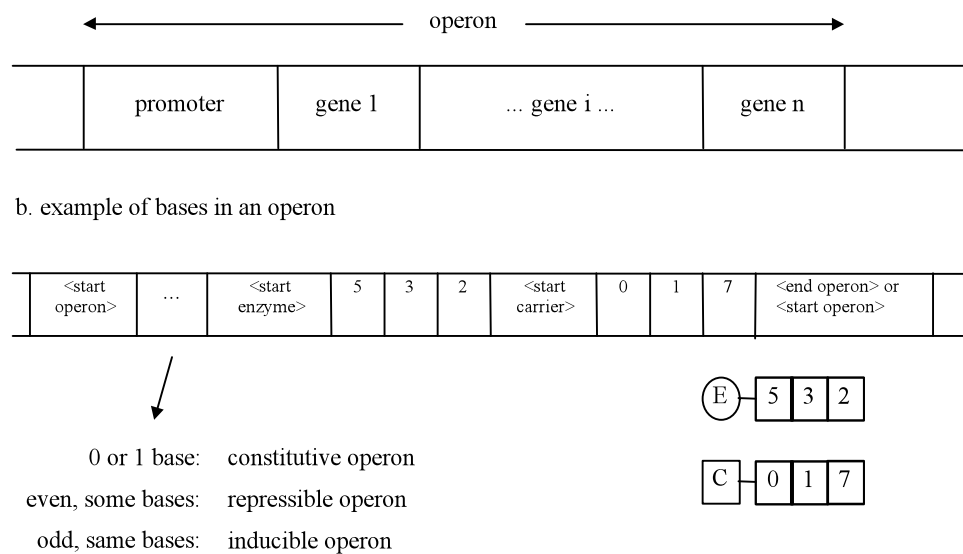


Figure 5.3: Operons. a: structure of an operon, b: bases in an example of an operon

Promoter Region

In our model, each operon consists of an optional promoter region followed by one or more templates for enzymes and/or carrier molecules. The *<start operon>* symbol marks the place on the genome where an operon begins. Bases in [0, 9] immediately following represent the promoter region of the operon. These bases specify which

chemicals can induce or repress expression of the genes following. The promoter region ends when a *<start operon>*, *<start enzyme>*, *<start carrier>* or *<end operon>* is read. The difference between this model and the operon model is that, for simplicity, we do not allow operons to have both an activator and repressor.

If there are no bases in the promoter region (that is, the *<start operon>* is followed directly by *<start enzyme>* or *<start carrier>*) or only one base, then the genes are constitutive: always turned on. When there are two or more bases in the promoter region, however, the operon can be regulated.

The first base of the region specifies whether the genes remain on until turned off (base is even), or are off until turned on (base is odd) under the influence of a suitable activator molecule. The bases in the promoter region following this first base are a specification of the kind of molecule that can switch the operon. The closer the bases in a molecule match this string of bases in the promoter, the more effectively it can regulate the genes.

Genes

Each gene in an operon is a template for either an enzyme molecule or a carrier molecule.

The template for enzyme molecules begins with the *<start enzyme>* base. Bases following specify the sequence of monomers comprising the enzyme. The activity and properties of the enzyme results from this sequence of monomers. Since enzymes are oriented molecules, their order is important. There must be at least two bases in an enzyme. This reflects their use in the model. Enzymes bind two substrate molecules into a product by simultaneously binding to the end of both substrates and bringing them close enough to bind into a product. This is an abstraction of how enzymes are thought to work in practice. Consequently, there must be at least two bases in the enzyme: one for the end of one substrate and one for the start of the second substrate. The enzyme specification ends when one of the special codes is encountered.

Templates for carrier molecules are similar, except they start with a *<start carrier>* base. As with enzymes, the bases that follow specify the shape of the carrier. Unlike enzymes, however, carriers can consist of only one base.

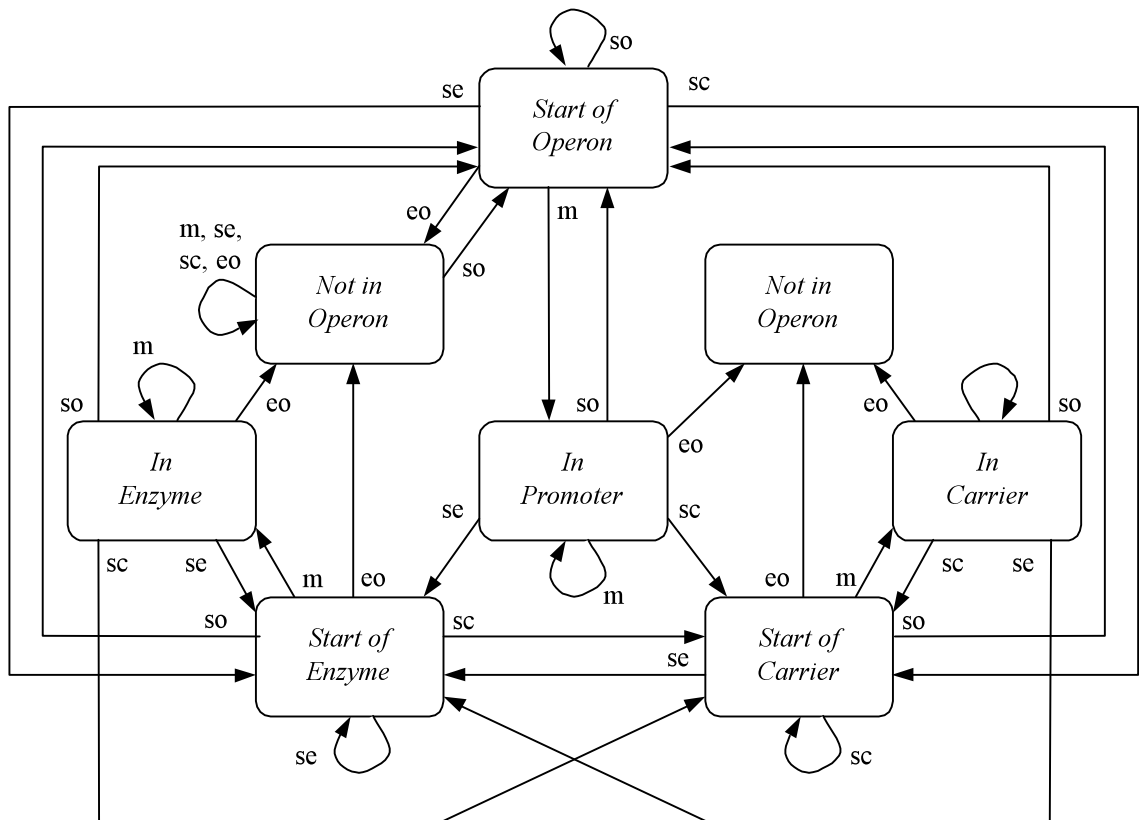
A formulation of the language used for one operon on our genome in context free grammar notation (Aho, Sethi and Ullman 1986, Lewis and Papadimitriou 1981) is given in Table 5.2. Although it is not completely regular, a deterministic finite automaton (figure 5.4) (Brookshear 1989, Lewis and Papadimitriou 1981, Aho, Sethi and Ullman 1986) can be used to parse the genome.

Deterministic finite state automata cannot parse non-regular languages. This machine is able to parse the non-regular gene language because it has extra functionality built into each state change.

Table 5.3 lists the actions taken at each state change. The name of each state in the finite state automata is given along with its meaning. Following this are three rows. The first row lists the action to be taken when making the transition into this state from another (different) state. The second row lists the action to be taken when moving into the same state, and the final row presents the action to be made when moving out of the state into another (different) state. For some of the actions an italicised description of the meaning of this kind of transition is given.

Table 5.2: Context free grammar definition of the gene language. It deals with only one operon not an entire genome.

$\langle \text{operon} \rangle \rightarrow \langle \text{start operon} \rangle \langle \text{operonbody} \rangle$
 $\langle \text{operonbody} \rangle \rightarrow \langle \text{promoter} \rangle \langle \text{genelist} \rangle$
 $\langle \text{operonbody} \rangle \rightarrow \langle \text{genelist} \rangle$
 $\langle \text{promoter} \rangle \rightarrow \langle \text{switchtype} \rangle$
 $\langle \text{promoter} \rangle \rightarrow \langle \text{switchtype} \rangle \langle \text{baselist} \rangle$
 $\langle \text{switchtype} \rangle \rightarrow \langle \text{repressible} \rangle \mid \langle \text{inducible} \rangle$
 $\langle \text{repressible} \rangle \rightarrow 0 \mid 2 \mid 4 \mid 6 \mid 8$
 $\langle \text{inducible} \rangle \rightarrow 1 \mid 3 \mid 5 \mid 7 \mid 9$
 $\langle \text{genelist} \rangle \rightarrow \langle \text{endofoperon} \rangle$
 $\langle \text{genelist} \rangle \rightarrow \langle \text{enzyme} \rangle \langle \text{genelist} \rangle$
 $\langle \text{genelist} \rangle \rightarrow \langle \text{carrier} \rangle \langle \text{genelist} \rangle$
 $\langle \text{enzyme} \rangle \rightarrow \langle \text{start enzyme} \rangle \langle \text{base} \rangle \langle \text{baselist} \rangle$
 $\langle \text{carrier} \rangle \rightarrow \langle \text{start carrier} \rangle \langle \text{baselist} \rangle$
 $\langle \text{baselist} \rangle \rightarrow \langle \text{base} \rangle \langle \text{baselist} \rangle$
 $\langle \text{baselist} \rangle \rightarrow \langle \text{base} \rangle$
 $\langle \text{base} \rangle \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
 $\langle \text{endofoperon} \rangle \rightarrow \langle \text{end operon} \rangle \mid \langle \text{start operon} \rangle$
 $\langle \text{start operon} \rangle \rightarrow a \mid b$
 $\langle \text{start enzyme} \rangle \rightarrow c \mid d$
 $\langle \text{start carrier} \rangle \rightarrow e$
 $\langle \text{end operon} \rangle \rightarrow f$



m: monomer $\in [0,9]$
 so: <start operon>
 se: <start enzyme>
 sc: <start carrier>
 eo: <end operon>

Figure 5.4: Deterministic Finite State Automata used to parse genome

The actions for each transition manipulate a number of variables that the parsing algorithm uses to accumulate information across the states. These require some explanation.

code is the nibble just read from the genome.

operon is the current operon that the parsing algorithm is accumulating. It consists of switching information and a list of genes. The switching information says whether the operon is constitutive, inducible or repressible. For inducible and repressible operons a string representing the molecular shape required to regulate the operon is recorded.

switch string is the regulation string for the promoter region in the current operon.

Once an operon has been parsed, it is added to a *list of operons*.

current enzyme string and *current carrier string* are the bases describing the shape of the gene we're currently parsing.

Table 5.3: Actions associated with each state change in the finite state machine.

Not in Operon	FSM is looking at a non coding region of the genome.
When entering	No action
Staying in same state	No action
When exiting	No action

Start of Operon	FSM has just read a <i><start operon></i> code and will look for a promoter region or genes.
When entering	No action
Staying in same state	<i>Ignore consecutive <start operon> codes by silently aborting the non-existent operons.</i> No action
When exiting	No action

Table 5.3 (cont'd): Actions associated with each state change in the finite state machine.

In Promoter	FSM is currently parsing the promoter region of an operon.
When entering	<p><i>We've just read the first base of the promoter region</i></p> <p>if code is even</p> <p style="padding-left: 40px;">mark the operon as repressible</p> <p>else</p> <p style="padding-left: 40px;">mark the operon as inducible</p>
Staying in same state	<p><i>We've just read the second or later base of the promoter region.</i></p> <p>Add code to the switch string we are accumulating</p>
When exiting	<p><i>The promoter has ended because either we're starting a gene (enzyme or carrier) or we've ended the operon.</i></p> <p>if code is <start enzyme> or code is <start carrier></p> <p style="padding-left: 40px;">if no bases were accumulated in the switch string</p> <p style="padding-left: 80px;">mark the operon as constitutive instead</p> <p style="padding-left: 40px;">Tell the operon what the switch string is.</p> <p style="padding-left: 40px;">Empty the promoter switch string for the next operon.</p> <p>else</p> <p style="padding-left: 40px;">Abort the operon and reinitialise the operon and switch string.</p>

Table 5.3 (cont'd): Actions associated with each state change in the finite state machine.

Start of Enzyme	FSM has just read a <i><start enzyme></i> code and expects the enzyme specification to follow.
When entering	No action
Staying in same state	<i>Ignore consecutive <start enzyme> codes by silently aborting the non-existent enzymes.</i> No action
When exiting	<i>The enzyme has ended before it really started. If we're also ending the operon save it away.</i> if code is <i><end operon></i> or code is <i><start operon></i> if there is at least one gene in the operon Add the operon to the list of operons for the genome. Reinitialise the operon .

In Enzyme	FSM is parsing an enzyme gene.
When entering	Add code to the current enzyme string
Staying in same state	Add code to the current enzyme string
When exiting	<i>We're finishing the enzyme because either we're starting another gene or ending the operon. Save the enzyme in the operon if it contains two or more bases.</i> if there are at least two bases in the current enzyme string Tell the operon about this enzyme Reinitialise the current enzyme string if code is <i><end operon></i> or code is <i><start operon></i> if there is at least one gene in the operon Add the operon to the list of operons for the genome. Reinitialise the operon .

Table 5.3 (cont'd): Actions associated with each state change in the finite state machine.

Start of Carrier	FSM has just read a <i><start carrier></i> code and expects the carrier specification to follow.
When entering	No action
Staying in same state	<i>Ignore consecutive <start carrier> codes by silently aborting the non-existent carriers.</i> No action
When exiting	<i>The carrier has ended before it really started. If we're also ending the operon save it away.</i> if code is <i><end operon></i> or code is <i><start operon></i> if there is at least one gene in the operon Add the operon to the list of operons for the genome. Reinitialise the operon .

In Carrier	FSM is parsing a carrier gene.
When entering	Add code to the current carrier string
Staying in same state	Add code to the current carrier string
When exiting	<i>We're finishing the carrier because either we're starting another gene or ending the operon.</i> Tell the operon about this carrier. Reinitialise the current carrier string . if code is <i><end operon></i> or code is <i><start operon></i> if there is at least one gene in the operon Add the operon to the list of operons for the genome. Reinitialise the operon .

5.6 Characterization of the Non-Coding Regions

Between operons are regions of the genome that do not encode proteins. In this model, these regions are invalid strings in the genome language (syntactically defective operons). They can become valid strings if a token (base) in this region or before mutates, or a crossover or inversion moves the invalid string into a coding region and the resulting string becomes legal. Semantically defective operons, that is operons that can be expressed but that contribute nothing or work against the metabolism, are valid operons, but will be selected out of the population.

We view non-coding regions as being 'almost' operons. They are either strings that can become operons after some modification or parts of operons that were lost after some past mutation. In the latter case, the non-coding region may have been useful to a previous form of the metabolism. If the current metabolism reverted to this previous form, perhaps as the result of an environmental change, the non-coding region may be easier recovered than reinvented. The environmental change we refer to may introduce a new chemical into the cell or remove an existing chemical species. This may well change the fine balance of reactions, and hence, the metabolism. The non-coding region, of course, would be recovered in the offspring since mutations occur only during breeding.

The more redundancy that exists in the genetic code, the closer the non-coding regions are to being fully-fledged operons, and the smaller the changes needed to transform a non-coding region to an operon.

5.7 Genetic Operators

The standard genetic operators found in genetic algorithms (Holland 1992) are applied to the genome. These are recombination, mutation and inversion. Typical rates used are given in Table 5.4.

Table 5.4: Typical rates for genetic operators

Strand length	400 bits
Crossover rate	0.04 (nibbles)
Mutation rate	0.005 (bits)
Inversion rate	0.15 (genomes)

The crossover rate is applicable to nibbles. This means we would expect around four crossover points on the genome. The mutation rate is calculated on the number of bits on the genome. Here we expect about two mutations per strand. Because one strand is calculated from the other, a mutation on one strand causes a mutation on the other strand. This means that the two expected mutations per strand result in four expected for the genome. The inversion rate relates to genomes. Fifteen percent of the time a genome will undergo inversion. When this occurs, two inversion points are chosen using a uniform distribution, as explained below.

Crossover and inversion occur usually only on nibble boundaries, although the alternative is assessed in section 10.3. The motivation here is to avoid unduly adding noise in the form of additional mutation into the genome when applying either of these operators. Crossover that didn't occur on nibble boundaries would be similar to the crossover coupled with mutations at the crossover point. A similar argument applies to inversion. Mutation, however, may affect any bit.

In our encoding scheme, where genes are not forced to begin at prescribed positions, we notice that crossover tends to force operons and genes to start at exactly the same nibble position over a population. That is, crossover forces alleles to be located at exactly the same nibble position on the genome.

Inversion

We use a non-standard algorithm to accomplish inversion. The algorithm is straightforward (figure 5.5) and relies on the double-strand encoding scheme described

in section 5.2. All changes are made to the first strand, since it is the only one stored. The sequence of bits in the second strand is derived from the first strand.

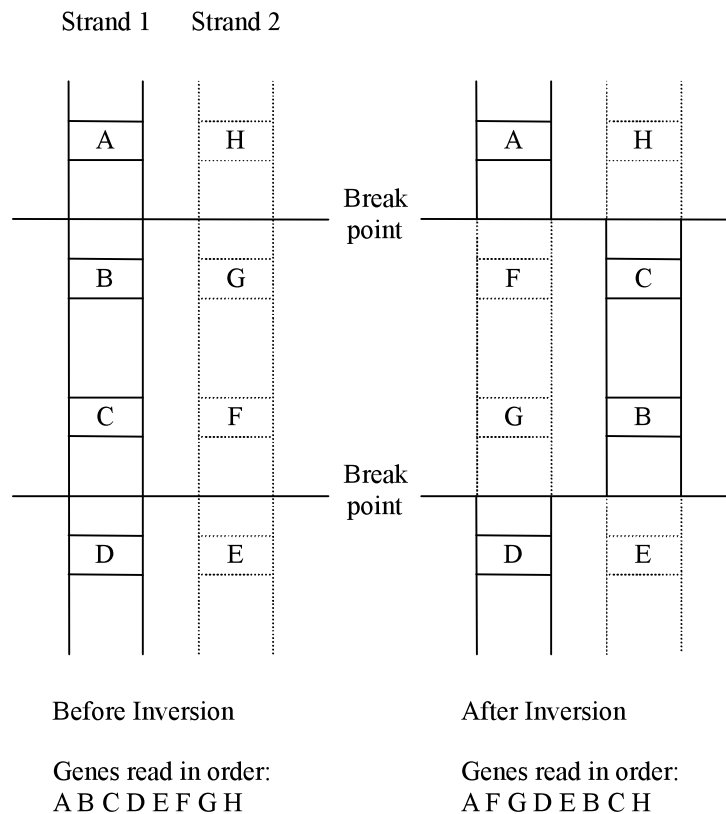


Figure 5.5: Effect of gene ordering by inversion

Our inversion algorithm reproduces the way inversion is believed to occur in Nature (Gardner, Simmons and Snustad 1991). Inversion occurs on a molecule of DNA when both strands break in two places. There are only two ways for the central region to be reconnected. It can connect the same way it broke off or rotated through 180° . In the latter case, the start of the break on the first strand joins with the end of the break on the second strand. These situations are illustrated in figure 5.5. Inversion results from the latter kind of break. The ordering of the region remains intact because the second strand is read in the reverse direction to that of the first strand.

When we want to invert a genome, we first choose start and end nibbles uniformly from the first strand. Next, we reverse the ordering of the bits in this region of the genome. Finally, we negate each bit in the region. Figure 5.6 illustrates this algorithm.

Any operons that are entirely inside the break region are moved intact from one strand to the other. From the point of view of gene expression and regulation, in which gene ordering is arbitrary, inversion leaves the genome unchanged. Inversion isn't completely benign, though, because genes (on either strand) may overlap the break points. These genes are scrambled and may become non-coding regions or move into a different operon. By the same process, though, non-coding regions may become fully-fledged genes again if they fall into coding regions. Inversion can move genes close to other related genes so that later recombination doesn't tend to separate them as easily. (Holland 1992)

	1	3	4	7	B	9	
Before inversion	0001	0011	0100	0111	1011	1001	stored strand
	1110	1100	1011	1000	0100	0110	computed
	7	3	D	1	2	6	
	break points						
Reverse	0001	1101	1110	0010	1100	1001	
Negate	0001	0010	0001	1101	0011	1001	
	1	2	1	D	3	9	
After inversion	0001	0010	0001	1101	0011	1001	
	1110	1101	1110	0010	1100	0110	
	7	B	7	4	3	6	

Figure 5.6: Inversion algorithm

Chapter 6

GENE EXPRESSION AND REGULATION

*Incy wincy spider went up the waterspout
Down came the rain and washed the spider out.
Out came the sun and dried up all the rain and
Incy wincy spider went up the spout again.
- Traditional*

In this chapter, we describe the model of gene expression and regulation in the cell simulation. The discussion proceeds from a conceptual model of biological transcription and translation to its formulation in the system of differential equations employed in our abstraction.

This chapter builds a model based on the brief description of biology presented in section 5.4. It also relies on the description of the genome in that chapter. Chapter 3 set out a high level description of the relationship between the genome and metabolism, but deferred the details to this chapter.

In order to simplify this chapter, only the results of mathematics needed to model the regulation of operons are used here. Their derivation is in appendix A.

Initially, we outline an abstraction of the biological processes of transcription, translation and gene regulation. Next, we describe an appropriate method of formalizing these abstractions in our cell simulation model. This formulation involves the addition of a new kind of chemical called a spider. The properties of spiders and the reactions in which they can participate will be described. This leads to the differential equations

required to model transcription and translation. Finally, we show how gene regulation can be accommodated into the abstraction and differential equations.

6.1 Abstracting Gene Expression and Regulation

Researchers do not fully understand the production of proteins in real cells. It is an extremely complex process involving many physical chemical processes. In our model we are interested mainly in the basic idea that information encoded in a genome guides the creation of related proteins and that this production can be regulated. We are not interested in every detail of how this is accomplished in biological cells. Some of the detail in biological cells is the result of their structure. One example of this is the fact that, since transcription and translation occur in different regions of the cell, they must be two separate processes. Because we do not model the fine structure of cells, we are not constrained by many of the details of cellular gene expression. The important point is that our model must adhere to the spirit of gene expression and regulation. That is, that information on the genome directs protein production and that the metabolism influences this guidance.

Our model, therefore, abstracts the processes of transcription and translation into one operation. Another principle we apply is that the process of protein production must be easily encoded in the notation of differential equations. Under these constraints, we devised a simple discrete model of gene expression and regulation.

6.2 Formulation of Gene Expression and Regulation

The basic premise of the discrete model of gene expression and regulation is to combine the functionality of the mRNA molecule with the ribosome machinery. This new entity moves along a genome transcribing bases (like mRNA) and, at the same time, translates the base into an amino acid and appends it to a growing protein molecule (like a ribosome). We call this abstract entity a ‘spider’. For the moment, consider spiders to be a new chemical species.

We imagine a population of spider molecules crawling along the genome. Each one adds a monomer to a protein string that it creates with each step. A spider can only attach to the genome at the promoter region of an operon. It crawls along the genome and drops off at the end of its operon. As it moves from one gene in its operon to the next, a spider finishes the protein for that gene and releases it into the metabolism. This also occurs at the end of the operon. When a spider falls off the end of an operon, it randomly joins another operon. Figure 6.1 illustrates the discrete model of gene expression.

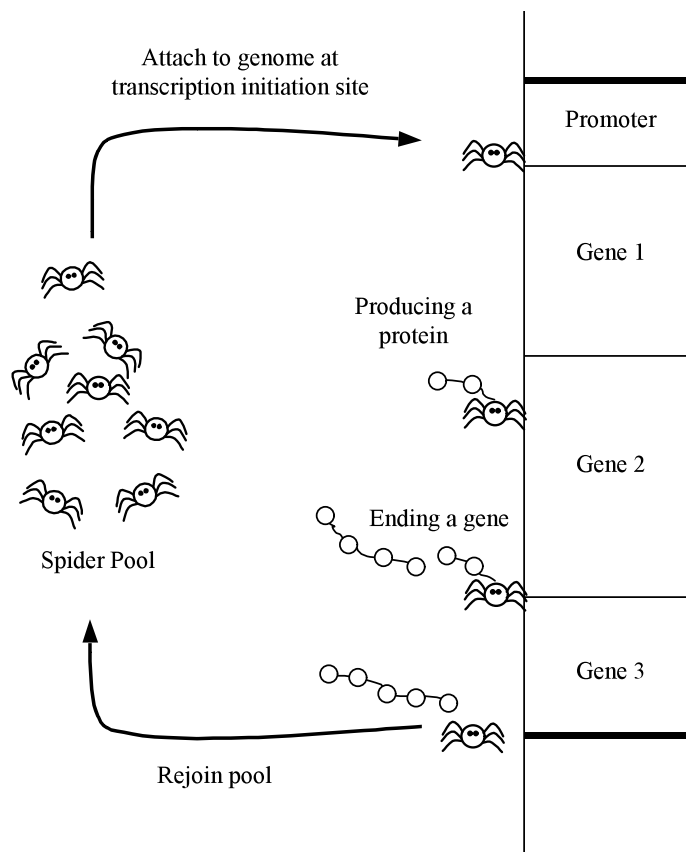


Figure 6.1 Discrete model of gene expression

Furthermore, a spider can only attach to an operon if the operon is ready for transcription (switched on). We model three kinds of operon: constitutive, repressible and inducible. Constitutive operons are always ready for expression. Spiders may attach to them at any time.

Repressible operons are switched on unless a ‘blocker’ molecule is bound to the promoter region of the operon, in which case the operon is turned off. Spiders can only attach to repressible operons if a blocker molecule doesn't happen to be bound to the promoter region at that time. Blocking molecules only stay attached to the genome for short periods. The more bases matching between the blocker and the promoter region, the stronger the bond between the blocker and the promoter, and the longer the blocker stays bound. Figure 6.2 illustrates repressible operons.

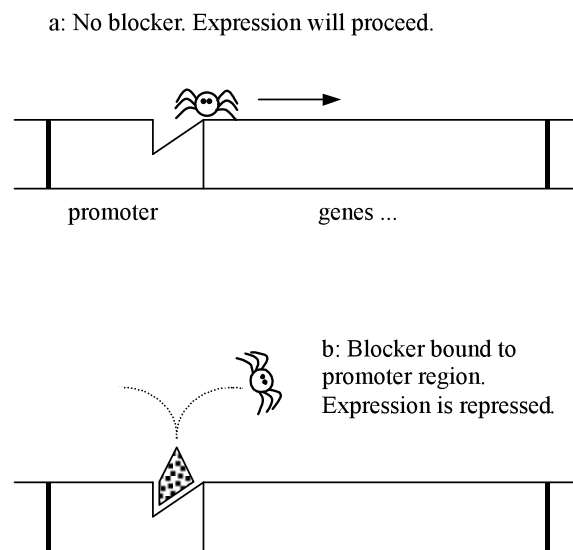
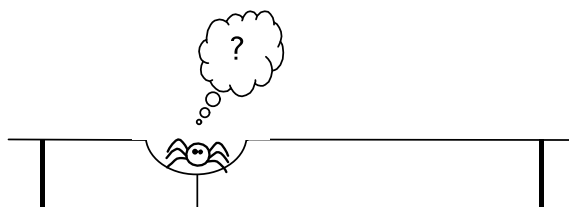


Figure 6.2 Repressible operons. a: Without blocker, b: blocker is bound

Inducible operons are turned off unless an activator molecule is bound to the promoter region, in which case the genes are switched on. Spiders can only attach to inducible operons if an activator molecule also happens to be bound to the promoter region at the same time. Inducible operons are illustrated in figure 6.3.

a: No activator. Expression cannot occur.



b: Activator bound to promoter region. Expression proceeds.

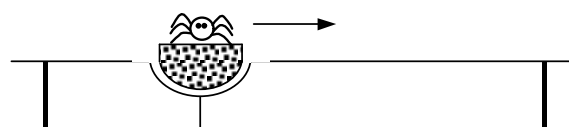


Figure 6.3 Inducible operons. a: Without activator, b: activator is bound

Spiders

Spiders are modelled in our system by a family of similarly shaped chemicals. Not all spiders join the genome or crawl along it at the same speed. The closer a spider matches the Platonic ideal of a spider chemical, the faster it can read the genome and add monomers to the trailing protein string. The standard (arbitrary) spider shape used most often in the simulations is *12312*.

The scheme we use to calculate the degree of similarity between a candidate spider molecule and the Platonic ideal is similar to algorithms used elsewhere in the simulation. For example, when calculating how well a chemical will regulate an operon.

In broad terms, we hold the candidate spider against the Platonic ideal in every possible position. In each position, we count the number of matching bases. The degree of

matching is the highest number of matching bases in a position divided by the number of bases in the Platonic ideal. If this fraction is greater than some threshold value (0.5) then the candidate chemical can act as a spider. The transcription rate for the candidate spider is equivalent to the degree of matching multiplied by the transcription rate of the Platonic ideal spider (typically 1.0×10^{-2}). As intimated in Table 3.2 only ordinary chemicals and enzymes can act as spiders. Spiders, then, may have more than one chemical property and use in a cell. Whenever molecules of these species are not expressing genes, they can act in their other capacity: as a catalyst or substrate. When they are bound to the genome, however, they must continue expression until the end of the operon.

6.3 Reactions and Differential Equations Required for Gene Expression

The discrete scheme outlined above, however, has a major flaw. It is discrete, whereas the rest of our model is continuous. We're interested in mass action, not the everyday trials and tribulations of specific molecules. To be encoded in a system of differential equations, this discrete abstraction must be made continuous. The way we have previously recast a discrete model of molecular interactions into a continuous one is by using reactions. These are easily encoded into differential equations.

Each step along the genome, then, is rewritten as an irreversible reaction with a spider molecule and monomer as reactants and a modified spider molecule and perhaps a protein (enzyme or carrier) as the product. The modified spider molecule becomes the reactant for the next base along the genome. Figure 6.4 gives the flavour of such a scheme for two operons, each containing one gene. The first reaction in each pathway has an additional parameter that gives the probability that the spider can bind to the gene at the current time. This parameter controls the regulation for the operon. This continuous model of gene expression using spiders gives the mean production of protein and activation of genes over time.

A number of assumptions are implied by phrasing gene expression in terms of these reaction graphs. Once a spider is committed to expressing an operon it cannot stop the gene expression until it reaches the end of the operon. As well, the gene expression is

perfect. No noise is introduced in either the transcription of the genome or the production of the protein. Cells may regulate mRNA or ribosomes separately (Alberts, Bray et al. 1994). As we have compressed the two functions into one, we do not have that freedom.

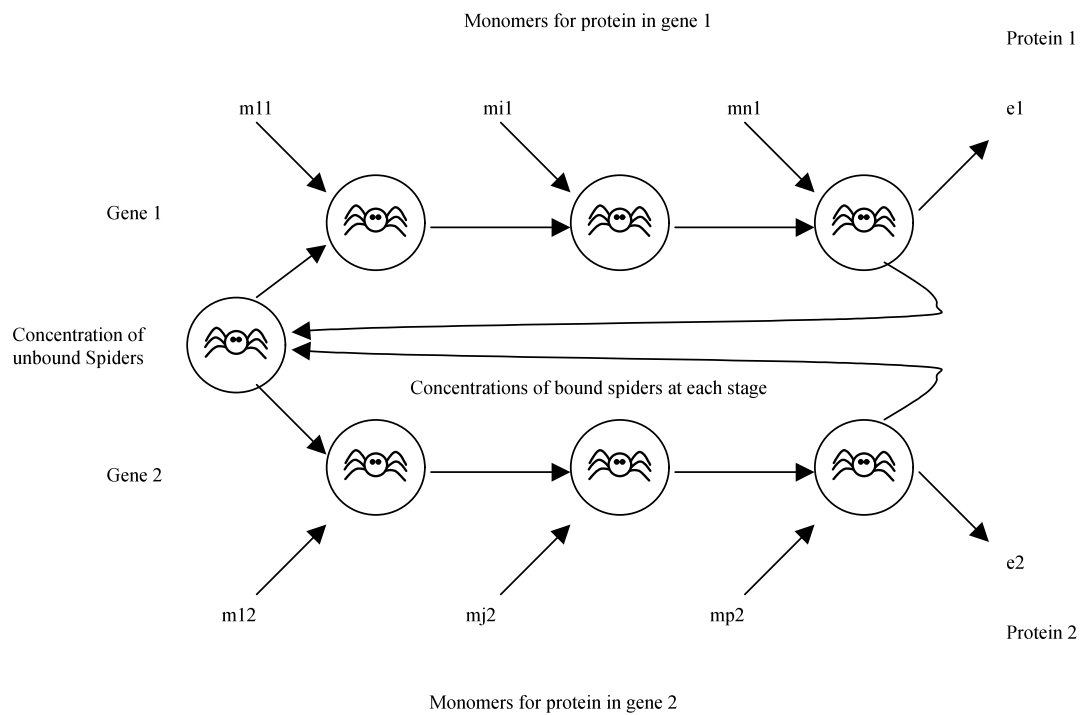


Figure 6.4: Gene expression reaction graph

Gene Expression Reactions

Two kinds of reactions are allowable in the expression reaction graph:

$$S + M \rightarrow S' \text{ and } S + M \rightarrow S' + P \tag{6-1}$$

The first reaction



models a reaction taking a spider molecule and a monomer in $[o0, o9]$ to produce a slightly different spider molecule. The substrate spider molecule may be a spider from the unbound pool or it may be a spider bound to the genome. The product spider molecule is a spider bound to the next position along the genome.

When this reaction is for the first base of the first gene of an operon, the rate constant needs to take into account gene regulation. In this case, the rate constant for the reaction is

$$G_i K_T \quad (6-3)$$

where G_i is the degree to which operon i is available for transcription (0 means the operon is turned off, 1 means it is fully turned on) and K_T is the transcription rate constant for the spider. The closer the spider resembles the ideal spider, the closer its transcription rate is to K_{TMAX} (typically 1.0×10^{-2} for the chemical *o12312*). Usually we seed the cell and environment with the spider chemical *o123*. G_i is described in the next section 6.4.

When this reaction is not the first of the operon, we choose either K_T or 1.0 as the rate constant. In the simulations of this thesis we choose 1.0 to make the cell simulations as fast as possible.

The other kind of reaction allowable in the transcription reaction graph is



This kind of reaction models a spider and a monomer combining to form another bound spider and a protein molecule (either an enzyme or carrier). Reaction (6-4) is used for the last base in a gene. When the first gene of an operon codes for a carrier molecule

with only one monomer, this reaction is also used. This is not the case for enzymes, of course, as they always contain at least two monomers. It is allowable for S' to be the unbound form of the spider. This case arises for the last base of the last gene of an operon, when the spider is returned to the unbound pool. The rate constant for reaction (6-4) is the same as above: $G_i K_T$ for the initial reaction of an operon or K_T or 1.0 for subsequent reactions.

The reaction graph for the expression of an operon must be replicated for each species of spider molecule in the cell. Different bound spider species are used. Bound spider species cannot act as enzymes or ordinary chemicals, so they are represented in the model as completely different species. They are the partially built enzyme and carrier molecules mentioned in Table 3.1 and given the name ' $pnnn$ ' where nnn is an increasing identification number. Every time we need to refer to new partially built protein, we use the next highest identification number. Partially built protein molecules cannot interact with the cell in any other way than reactions (6-2) and (6-4). In particular, they cannot diffuse through the cell membrane.

Differential Equations

Derivation of the terms in the differential equations for species in reactions (6-2) and (6-4) are straightforward. They follow directly from analytical chemistry.

For both reactions (6-2) and (6-4), the term

$$-kSM \tag{6-5}$$

is appended to the differential equations for the substrate spider species and appropriate monomer species where k is the rate constant of the transcription reaction (discussed above), S is the concentration of the substrate spider species and M is the concentration of the monomer species.

6.4 Gene Regulation

To model the switching of genes, we calculate the probability that a spider may attach to the start of an operon at a given time. This is the variable G_i mentioned above. G_i is another variable in the system of differential equations. The derivative of the function to calculate the probability is added to the system of differential equations.

The probability that a constitutive operon will accept a spider is always 1. Inducible and repressible operons only accept spiders when a switch chemical is bound to the promoter region or not (respectively). For inducible operons, the probability of accepting a spider is the same as the probability of an activator chemical being bound at that time to the promoter region. In the case of repressible operons, the probability of transcription is one minus the probability of a blocker being bound to the promoter region.

Determination of the probability that one of a number of chemical species able to regulate the operon is bound to the promoter region is more difficult. We calculate it based on a combination of the probabilities of a particular chemical species binding to the promoter, which is simpler to quantify. The results of these calculations are given below, but details are provided in Appendix A.

The probability that one molecule of a particular chemical species (s_i) is bound to the promoter region is

$$\rho_i = 1 - e^{-K_i s_i} \quad (6-6)$$

where s_i is the concentration of the chemical able to regulate the gene and

$$K_i = \frac{K}{(1 - q_i) e^{-n_i b}} \quad (6-7)$$

q_i is the probability that switching chemical s_i will not immediately bind to the promoter region: the probability of a bounce. This is calculated by

$$q_i = \frac{\text{number of positions with no matches}}{\text{total number of possible positions}} \quad (6-8)$$

We look at all the possible ways the regulating chemical can bind to the promoter region and count all the positions where there are no bonds (matches) between the chemical and the promoter region.

The other part of the denominator of K_i specifies how long the chemical will bind to the promoter region. This time derives from the Boltzmann distribution and depends on the number of bonds between the chemical and the promoter region and the strength of each bond. b is the relative strength of a bond (typically 0.25) and n_i is the average number of bonds between the regulating chemical and switching region.

$$n_i = \frac{\text{total number of bonds over all positions}}{\text{number of positions}} \quad (6-9)$$

K is a constant used to calibrate the concentration of a switching chemical with the probability that the chemical will be bound to the promoter region. Typically, we use the value 1.0×10^3 .

The probability of a chemical of one species binding to the promoter region is used to determine the probability that a molecule of one of n chemical species is bound to the promoter region. The derivation of this is in Appendix A and the result below.

$$\hat{\rho}_n = \frac{\rho_1 + (1 - \rho_1)I(n)}{1 + (1 - \rho_1)I(n)}, \quad (6-10)$$

$$I(n) = \sum_{i=2}^n \frac{\rho_i}{1 - \rho_i}$$

where n is the number of species of switching chemical.

and ρ_i is the probability that a chemical of species s_i will bind to the promoter region (assuming no other competing species).

We need a $\hat{\rho}_i$ variable for each operon on the gene and its derivative to add to the system of differential equations. The derivative is given below, and its derivation is in Appendix A.

$$\frac{d\hat{\rho}_n}{dt} = \left(\frac{1 - \rho_1}{1 + (1 - \rho_1)I(n)} \right)^2 \sum_{i=1}^n \left(\frac{1}{(1 - \rho_i)^2} \frac{d\rho_i}{dt} \right) \quad (6-11)$$

$$\frac{d\rho_i}{dt} = K_i e^{-K_i s_i} \frac{ds_i}{dt} \quad (6-12)$$

This concludes the mathematics necessary for gene expression.

Chapter 7

MODEL OF INTERACTION WITH ENVIRONMENT

*Two households, both alike in dignity,
In fair Verona, where we lay our scene,
From ancient grudge break to new mutiny,
Where civil blood makes civil hands unclean.
- William Shakespeare, "Romeo and Juliet"*

In this chapter, we discuss the relationship between the cell and its environment. There are three main aspects to this interdependence: the environment itself, growth of the cell, and diffusion of molecules through the cell membrane. The latter two relationships are the final metabolic processes listed in section 4.1.

This chapter concludes the specification of the cell model started in chapter 4. Chapter 8 shows how the cell model fits into a population of cells and the genetic algorithm.

Initially we describe the environment the cell inhabits. Next we introduce the notion of a membrane containing the cell and mechanisms for regulating cell size. Finally, we describe two methods that the cell can use to communicate with its environment. There is a slow background diffusion of chemicals between the environment and cell and a much faster facilitated diffusion. Facilitated diffusion requires the cell to produce proteins that can ferry molecules across the membrane. The cell has little control over the background diffusion. It has more influence over the facilitated diffusion because it may evolve tailor-made carrier proteins to transport particular molecules.

7.1 Environment

The environment around the cell is modelled as a very large (compared to the cell) well-stirred vessel. That is, the diffusion rate of chemical species from the vicinity of the (exterior) cell wall, as well as stirring, is assumed to be sufficient for the concentrations in that region to equal environmental levels. It contains a small number of chemical species relative to the cell. For reasonably stable, ie, viable, cells, any diffusion out or in should be slight to moderate, yielding a quasi-stable cell. Otherwise, cell death would be extremely likely.

These species are simple compared with those in the cell. There are no enzymes, carriers or partially expressed proteins. We assume these polymers are too large to diffuse through the membrane out of the cell. Only ‘ordinary’ chemicals (that is, those that are not enzymes or carriers as defined in section 3.3) appear in the environment. Because the environment has no enzymes and because spontaneous reactions are not modelled, no reactions occur there. The concentration of all chemical species outside the cell, therefore, remains constant.

More explicitly, chemical species in the environment are held at fixed concentrations. Any molecules diffusing out of the cell are assumed lost in the large environment. Consequently, no differential equations are required to model the chemical interactions in the environment.

7.2 Cell Growth: Modifying the Cell Membrane

Changes in cell size are accomplished by modification of the cell membrane. An increase in the number of cell membrane molecules leads to growth of the cell wall surface area (and hence volume). Conversely, a reduction of the number of cell wall molecules implies a diminution of the cell. Cell membrane molecules are treated differently to other molecules in the cell. They are not modelled explicitly as chemical species, but rather as a raw number of an unspecified (family of) chemical species. This design decision is discussed later in this section.

An important attribute of real cells is the mechanisms they use to cope with inorganic ions (Alberts, Bray et al. 1994). There tend to be more large molecules such as proteins inside cells than in the environment. Cells also contain many smaller charged molecules. Both these small charged molecules and the macromolecules attract inorganic ions. So, there tends to be more inorganic ions inside the cell than just outside it. The process of osmosis balances the concentrations of ions in two neighbouring solutions by moving water into the solution with the higher concentration. This means that water flows into the cell, possibly breaking the membrane and bursting the cell. Animal cells and bacteria continually pump ions out of the cell. Plant cells have hard cell walls that can cope with the osmotic pressure and protozoa push water out from special vacuoles. Inorganic ions are not modelled in our artificial cells, so we will add a simplifying assumption. The cell membrane is able to equalise the water pressure in the cell by some means not modelled. The cell is kept as a plump sphere.

Calculating the Size of the Cell

We assume that the cell forms a sphere, filled with as many water molecules as possible given the available surface area. Biological cells achieve the same plumpness by more sophisticated mechanisms. We do not, for example, consider the effect of ions in the cell or give the cell any internal structure from microtubules. The surface area of our cell is a function of the number of cell wall molecules in the cell.

$$A = \frac{8\pi^2 r^2 N_M}{9(1 + \sqrt{2/3})} \quad (7-1)$$

where r is the radius of a cell membrane molecule (assumed spherical) and N_M is the number of cell membrane molecules “in” the cell. The derivation of (7-1) is in Appendix B. Using (7-1), the number of water molecules able to fit in a cell containing N_M cell membrane molecules can be calculated.

$$N_{WC} = \frac{N_A}{A_W} 10^6 \frac{4\pi}{3} \left(\frac{2\pi r^2 N_M}{9(1 + \sqrt{2/3})} \right)^{3/2} \quad (7-2)$$

where N_A is Avogadro's number (6.0220×10^{23}) and A_W is the atomic mass of a water molecule (18.0152 daltons). Derivation of (7-2) is given in Appendix B.

Effects of Changing the Size of the Cell

Changes in cell surface area directly determine the number of water molecules held in the cell. Because all chemical concentrations in the cell are a function of the number of water molecules, changes in cell size occasion small modifications to the concentrations of chemicals in the cell. Usually the change in cell size over one time step is slight and the difference in number of water molecules relatively small. Modifications to chemical concentrations as the result of a large change in the number of water molecules over a time step are required very rarely. The correction factor applied to each concentration variable in the cell is

$$\frac{N'_{WC}}{N''_{WC}} \quad (7-3)$$

where N'_{WC} is the number of water molecules in the cell in the previous time step and N''_{WC} is the number of water molecules in the cell in the new time step. This factor effectively converts the concentration to a raw number of molecules, then back to a concentration.

Changes in the number of water molecules in the cell large enough to warrant application of factor (7-3) are rare events. This is because there are very many water molecules in the cell, typically around 2.7×10^{14} . We calculated this number from the volume of a typical prokaryote cell.

When factor (7-3) is used to correct the chemical concentrations in the cell, it often augurs catastrophe to the cell. If a cell can manipulate the number of cell membrane molecules to such a large extent, it is usually unable to control this process. Decreasing the cell size hastens chemical reactions. This may increase the shrinkage of the cell.

Increasing the cell size, conversely, slows metabolic reactions. This may make it harder for the cell to halt the increase. This topic is discussed in more detail in chapter 9.

The Nature of Cell Wall Molecules

Cell wall molecules are not modelled as separate chemical species in the cell. This is because chemical species, in our model, are represented with a concentration. Concentrations of chemicals are specified as a function of the number of water molecules in the cell (see section 3.3). However, from equation (7-2) above, the number of water molecules is a function of the number of cell wall molecules (via cell volume). To resolve this circular definition, we model the cell membrane as a raw number of molecules.

Change in the number of cell membrane molecules is accomplished using two families of chemical species. The presence of one family increases the number of cell wall molecules, whilst the presence of chemicals from the other family decreases it. The cell must maintain the correct balance of these chemicals under environmental pressure to ensure the stability of its membrane. We use a similar matching scheme to that already described when choosing spider molecules (section 6.2) to determine whether a particular species of chemical belongs to either family of chemicals. Membrane building and breaking molecules are built from metabolic reactions or diffuse into the cell from the environment.

The differential equation governing the number of membrane molecules “in” the cell is specified as:

$$\frac{dN_M}{dt} = N_{WC} \left(k_1 \sum (\text{builder concs}) - k_2 \sum (\text{breaker concs}) \right) \quad (7-4)$$

Typically we use $k_1 = k_2 = 1.0 \times 10^{-6}$. These values are arbitrary but plausible. Their relationship with other parameters such as the metabolic reaction rates is more important than their exact values.

Note that N_{WC} is a function of N_M . Here, however, we treat its last computed value as a constant or dN_M/dt is small. N_{WC} multiplied to the concentrations gives the number of builder and breaker molecules. Multiplying by k_1 gives the number of new membrane molecules. Similarly, multiplication by k_2 gives the number of membrane molecules destroyed.

We set the archetypal builder and breaker molecules as arbitrary sequences: usually *8441* for the builders and *0307* for the breakers. As well, we usually seed the initial cells and the environment with the chemicals *o44* and *o30*. This allows the cell to find new builders and breakers easily.

7.3 Transmembrane Diffusion

We model two kinds of diffusion in the cell: slow background diffusion and a faster, more specific facilitated diffusion.

The slow background diffusion has rate inversely proportional to the length of the diffusing polymer and models the transfer of small molecules directly through the membrane. Faster facilitated diffusion models assisted transport of specific molecules through the membrane. This can be visualised as using carrier molecules that act as chaperones through the membrane, or using transmembrane portals through which molecules of a particular shape can traverse.

The diffusion of only ordinary chemicals through the semipermeable membrane is modelled. We assume that enzymes, carriers and chemicals bound in reactions are too large to diffuse through the cell wall.

Background Diffusion

Background diffusion models the transfer of small molecules through the cell membrane. We assume this is the result of chemicals jostling against the wall and forcing a path between the cell membrane molecules, or finding gaps.

Osmotic pressure (Alberts, Bray et al. 1994) suggests that this diffusion is proportional to the difference between the concentration of the molecule inside the cell and outside in the environment.

$$\frac{dx_i}{dt} = \dots - K(x_{i\text{IN}} - x_{i\text{OUT}}) \quad (7-5)$$

where $x_{i\text{IN}}$ is the concentration of x_i inside the cell (ie, x_i) and $x_{i\text{OUT}}$ is the concentration of chemical x_i in the environment. K is a diffusion rate constant that is inversely proportional to the linear dimension of the diffusing molecule, which we model as the cube of the number of monomers comprising the polymer.

$$K \propto \frac{1}{l_i^3} \quad (7-6)$$

As well, the rate of diffusion is directly proportional to A , the surface area of the cell (7-1). This multiplier is derived in Appendix B.

$$K \propto A = \frac{8\pi^2 r^2 N_M}{9(1 + \sqrt{2/3})} \quad (7-7)$$

where r is the radius of a cell membrane molecule (typically 4.4×10^{-9} m) and N_M is the number of cell wall molecules in the cell. This term, unlike (7-6) is the same for all species of diffusing chemical.

Background diffusion, then, adds the following term to the differential equation of the diffusing chemical species.

$$\frac{dx_i}{dt} = \dots - K_C \frac{8\pi^2 r^2 N_M}{9(1 + \sqrt{2/3})l_i^3} (x_{i\text{IN}} - x_{i\text{OUT}}) \quad (7-8)$$

where K_C is a global parameter used to calibrate the model. Usually K_C has the value 2.0×10^4 . Again, the actual value of this parameter is arbitrary, but its relationship with other parameters is important. We do not ask questions of the model that are answered directly by the parameter K_C (for example). Our model is not predictive. Instead, we ask questions about the behaviour of the cell. This behaviour is different for different values of the diffusion rate, but it is still interesting behaviour within the context of the parameters. As well, evolution of the cells must react to a given set of parameters.

Facilitated Diffusion

The model for facilitated diffusion is more complex and uses reaction graphs. By facilitated diffusion, we mean the transport of a molecule across the cell membrane using another chemical.

Two models are used to explain facilitated diffusion. (Alberts, Bray et al. 1994) One model uses a carrier chemical that binds to the permeant, makes some conformational changes that moves the permeant through the membrane and unbinds on the other side. The other model uses a protein that acts as a pore in the cell membrane, allowing only permeants of a particular shape and chemical composition access. Both models are able to transport chemicals through the membrane in the “downhill” direction. By this, we mean from the side with the higher concentration of permeant to the side with the lower concentration. Carriers can also transport permeants through the membrane in the uphill direction using power sources such as ATP. Only downhill (passive) transport appears in our model.

Cells use facilitated diffusion to move ions and small polar molecules such as sugars or amino acids through the cell membrane. The membrane is otherwise impervious to these molecules. Currently, in our model, we allow the analogue of amino acids (monomers) to travel freely through the membrane. We felt that not doing so would have made solution of an environment too difficult for the cells. In any case, we are interested more in the control of the cell than in how the cell accumulates raw materials. A more selective membrane can easily be modelled by significantly reducing K , the background diffusion rate constant in (7-8).

Since we are not interested in active (uphill) transport, and both carriers and channels are implemented in our model using the same reaction graphs, the actual means of transport does not concern us.

Our implementation of facilitated diffusion relies on the treatment given in Segel Ch. 6 (Segel 1980b, Rubinow 1980). However, our formulae are modified significantly to fit into the system of differential equations used in our model.

The basic model of binding and unbinding permeant and carrier molecules and the transfer of the complex through the cell membrane uses a graph of three reversible reactions:



where k_+ is the rate constant determining the speed with which the permeant and carrier molecules bind into a complex. Similarly, k_- is the backward binding rate constant. k_d is the speed of diffusion through the membrane. Although, in (7-9) above, k_d appears to be a rate constant, it is actually variable. As we discuss later, k_d is determined from the interaction of the membrane surface area and flux per unit area.

p_e is the (concentration of) permeant outside the cell and p_i is the permeant inside. Similarly, c_e is the carrier on the outer side of the membrane and c_i is the carrier on the inner side. b_e is bound carrier permeant complex on the outer side of the membrane and b_i is the concentration of the complex on the inside.

One assumption we make in our formulation using reactions is that diffusion is an all-or-nothing affair. The transport occurs in one indivisible step. It can not be reversed once it has begun.

The terms to be added to the system of differential equations are:

$$\frac{dp_e}{dt} = \frac{dc_e}{dt} = k_-b_e - k_+p_e c_e \quad (7-10)$$

$$\frac{db_e}{dt} = k_+p_e c_e - k_-b_e + k_d(b_i - b_e) \quad (7-11)$$

$$\frac{db_i}{dt} = k_+p_i c_i - k_-b_i - k_d(b_i - b_e) \quad (7-12)$$

$$\frac{dp_i}{dt} = \frac{dc_i}{dt} = k_-b_i - k_+p_i c_i \quad (7-13)$$

For k_+ we typically use the value 1.0×10^{-5} , and for k_- the value 1.0×10^{-1} .

The rate of diffusion k_d is treated differently. Unlike all other reaction rates, it is not a constant. This reflects the fact that we are not modelling a reaction, but the movement of a bound molecule. We use the surface area of the cell multiplied by the flux per unit area. The flux per unit area changes with the difference of the concentrations of the permeant inside and outside the cell.

$$k_d = A \times J \quad (7-14)$$

where A is the surface area of the cell (derived in Appendix B)

$$A = \frac{8\pi^2 r^2 N_M}{9(1 + \sqrt{2/3})} \quad (7-15)$$

and J is the flux per unit area. This comes from the integrated form of Fick's law of diffusion given in Rubinow (1980).

$$J = \frac{D_b}{L} (p_e - p_i) \quad (7-16)$$

where D_b is diffusion coefficient of the bound carrier molecule. Typically, we use the value 1.0×10^8 . We assume that this is the same for all species of bound carrier molecule. Another assumption we make is that the size of the bound complex is similar to the size of the carrier molecule on its own. That is, that the permeant molecule is significantly smaller than the carrier molecule.

L is the distance to be diffused. This distance is the width of the membrane.

The final rate constant, then, is

$$k_d = \frac{4\pi^2 r N_M D_b}{9(1 + \sqrt{2/3})^2} (p_e - p_i) \quad (7-17)$$

Chapter 8

MODEL OF GENETIC ALGORITHM

*They sought it with thimbles, they sought it with care;
They pursued it with forks and hope;
They threatened its life with a railway-share;
They charmed it with smiles and soap.
- Lewis Carroll, "The Hunting of the Snark"*

We are concerned with the genetic algorithm (GA) shell in this chapter. The main body of the chapter consists of a discussion of the genetic algorithm: differences to the standard GA, formulation of the fitness function and a description of the details of experimental runs.

Chapters 3 through 7 described the model of an individual cell and its environment. Here, we examine the next level of complexity in the model. Chapters 9 and 10 present and analyse the experiments performed on the model.

Initially, in this chapter, we describe the genetic algorithm used to evolve the cells. There is a variety of differences between our genetic algorithm and that commonly used. Contrary to many genetic algorithms we do not employ fixed generations. As well, the method we use to replace members of the population with children varies from the norm. Next, we describe the fitness function used to quantify the performance of individual cells. Finally, we explore some side-effects of running the cell simulations over a network of asynchronous processors.

8.1 The Genetic Algorithm Shell

A genetic algorithm (Holland 1992, Goldberg 1989) evolves a population of cell simulations. Each simulation represents an individual phenotype. This algorithm runs each simulation over a network of processors and breeds the most successful simulations (cells) together, replacing one of the finished simulations in the population with the new child using one of a variety of methods. Each cell in the population performs the same task in a separate environment. The choice of running simulations over a network of asynchronous processors forces us to employ a particular scheme for the management of the genome population.

In section 2.5, we reviewed the fundamentals of genetic algorithms. Here we will describe the specialisations made to the basic genetic algorithm for our model.

Building a Phenotype from a Genotype

The model of individual cells has been revealed in previous chapters. A few comments are necessary to show exactly how a phenotype is built from a genotype.

As mentioned in section 3.1, our phenotype consists of a system of differential equations and a matching set of real valued variables and initialisations. The genotype, as described in section 5.2 is a double strand bit encoding.

The first step in building the phenotype is to parse the double strand bit genome into an operon parse tree.

Chemical species are then read from the mother cell. One of the two parents is randomly designated as the mother cell. Not all chemicals are given to the child. Partially built proteins, and chemicals caught in the mother's cell wall are not bequeathed. Partial enzymes and carriers are assumed an integral part of the mother's genome and they along with the partly diffused chemicals are assumed lost in the process of meiosis. Chemicals bound to enzymes as a part of the metabolic process, however, are passed on.

A reaction graph consisting of metabolic, transcription and facilitated diffusion reactions is then determined from matching the operon parse tree with the chemical species.

Next, the set of variables is determined. The number of variables of each type is given in table 8.1. Note that the number of variables changes as the simulation runs as new chemicals are found.

Table 8.1: Number of instances of each kind of variable

Type of variable	Number of instances
Ordinary chemical and enzyme	Two for each species (concentration and cumulative bound concentration)
Partially expressed protein	The number of spider species times the number of partially expressed protein required to express the genome. This latter value is approx. equal to the number of bases in the genome that code for a monomer in a protein.
Activation of operon	One for each operon on the genome.
Carrier protein	Two for each species. One for the concentration inside the cell and one for the concentration on the outer edge of the membrane.
Bound carrier / permeant complex	For each carrier species there will be a variable for each permeant it can carry across the membrane. One for the concentration of the bound complex / permeant inside the cell and one for the concentration of the bound carrier / permeant complex at the outer edge of the membrane.
Cell membrane molecules	One variable

Following this, terms in the differential equations are calculated from the reaction graph (table 8.2). The variables are now initialised. In most cases, initial chemical concentrations are set to the final chemical concentrations in the mother cell.

Table 8.2: Differential equations and terms that apply to each variable.

Differential Equation or term.	Variable(s) it applies to	Notes
Metabolic reaction (4-7 and 4-8)	Ordinary chemical and enzyme variables	One for the x_i (4-7) and one for \bar{x}_i (4-8)
Protein degradation (4-10) – of the protein	Enzyme and carrier variables	
Protein degradation (4-10) – returned to monomers	Ordinary chemicals that are also monomers (o1, o2, ... o9)	Depends on whether the protein contained the monomer in question.
Gene expression (6-2) – $S + M \rightarrow S'$	Spider chemicals, partially expressed proteins and monomers	This term is added to partially expressed proteins and subtracted from the others
Gene expression (6-4) - $S + M \rightarrow S' + P$	Spider chemicals, partially expressed proteins, monomers and proteins (either enzymes or carriers)	This term is added to partially expressed proteins (and spiders if the reaction is for the last base in the gene) and proteins. It is subtracted from the others.
Operon activation (6-11 and 6-12)	Operon variable	This is the DE for the operon variable.
Number of water molecules DE	Number of water molecules	This is the DE for the change in the number of water molecules
Background diffusion	Ordinary	This term is added to the DE

(7-8)	chemicals	of all affected (x_i NOT \bar{x}_i)
Facilitated diffusion – (7-10)	Variable for the concentration of carrier molecules on the outer edge of the membrane.	This term is added.
Facilitated diffusion – (7-11)	Variable for the concentration of bound carrier molecules at the outer edge of the membrane.	This term is added.
Facilitated diffusion – (7-12)	Variable for the concentration of bound carrier molecules inside the cell.	The term is added.
Facilitated diffusion – (7-13)	Variable for the concentration of permeant or carrier inside the cell.	The term is added to the appropriate DE. For the permeant, this is the x_i variable not \bar{x}_i .

Fitness Function Used to Score Phenotypes

A fitness function is used for each cell to give a measure of how well the cells adapted to living in each environment. Tautologically, fitter individuals yield higher fitness scores.

We employ six simple fitness metrics, each in $[0, 1]$. The final fitness value is the sum of the six metrics, plus the sum of all products of pairs of metrics, then triples, quadruples and quintuples of metrics plus all six metrics multiplied together.

Constructing the fitness in this way allows members to get a progressively higher fitness

value as each metric is optimised. It also allows us to reward particular combinations of metrics if we think they are important by applying a constant multiplier to that term. Finally, it allows us to have fitness components that conflict with one another.

The six components to fitness were derived with the intention they would form a canonical set. Qualitatively they are:

- Change in the cell's volume over the course of its lifetime relative to a target value
- The “time” the cell lived
- How closely correlated the genome switching regions were to the membrane builder and breaker chemicals
- The complexity of the metabolic reaction graph
- The number of membrane building and breaking species available to the cell
- The number of differential equations in the cell

Volume Metric

The primary fitness metric measures the change in volume over the cell's life. We wish to evolve cells that stay the same size, or grow a small amount. Given this constraint, we fashioned a function similar to that graphed in figure 8.1 to give the component of fitness derived from the change in volume observed in the cell. The actual function used is much narrower, but has been stretched out in figure 8.1 to show the general shape. The function is based on a sum of three exponential curves centred on zero, each narrower than the last. The values for the constant multipliers are typically very large in the simulation, and the curves therefore very narrow. If we had used only one exponential it would have appeared as a spike and hill climbing would have been difficult. Similarly, a very wide exponential would have been useless. A compromise of the sum of three exponential functions allows hill climbing to occur. We wish to allow cells to grow a little, but be punished for shrinkage. The simplest way to achieve this is to use a piecewise fitness curve. Constants for the exponential curves where the change in volume is positive are larger than when the change in volume is negative, thereby making the fitness more tolerant of slight growth.

This function is defined as

$$\begin{aligned}
 v &= \frac{e^{-k_{\text{FlatShrink}}vc^2} + e^{-k_{\text{BumpyShrink}}vc^2} + e^{-k_{\text{PointyShrink}}vc^2}}{3}, & \text{for } vc \leq 0 \\
 &= \frac{e^{-k_{\text{FlatGrow}}vc^2} + e^{-k_{\text{BumpyGrow}}vc^2} + e^{-k_{\text{PointyGrow}}vc^2}}{3}, & \text{for } vc > 0
 \end{aligned} \tag{8-1}$$

where $vc = \frac{\text{actual change in volume}}{(\text{estimated change in volume without reactions}) \log_{10}(\text{MIN}(T_{\text{cell}}, T_{\text{MAX}}))}$

and where T_{cell} is the time the cell lived and T_{MAX} is the maximum time the simulation is run.

The multipliers used in the simulation are:

$k_{\text{FlatShrink}}$	7.8×10^{-8}
$k_{\text{BumpyShrink}}$	7.8×10^{-7}
$k_{\text{PointyShrink}}$	7.8×10^{-6}
k_{FlatGrow}	7.8×10^{-8}
$k_{\text{BumpyGrow}}$	7.8×10^{-6}
$k_{\text{PointyGrow}}$	7.8×10^{-5}

Division by three scales the fitness component back to [0, 1].

The input into the fitness component function requires some explanation. Rather than use the simple change in volume of the cell, we normalise this value by the change that would occur from background diffusion if the cell had taken no metabolic action. See appendix C for derivation. The longer a cell lives, the further these changes in volume become, so we also normalise by the logarithm of the time the cell lived or the maximum time the simulation was run. We divide by the lesser time so as not to give an unfair advantage to cells that have virtually no metabolic activity. These cells have smooth curves so the numerical integration algorithm can proceed in very large time steps. Since simulations are ended when the last time calculated is greater than a maximum time these cells can live to a very long time after the maximum time (as long

as it is one increment) and score highly. Dividing by the maximum time rather than the actual lifetime selects against these cells.

The motivation behind normalising the change in cell's volume by an estimation of the change from diffusion alone is to select against cells that come from long lineages. We call these cells “aristocrat's sons”. When the cell's task is difficult, and takes many generations to accomplish, cells coming from a long lineage inherit initial concentrations that are much closer to the environment. That is, initial concentrations of membrane building and breaking chemicals that are similar to the environment. Consequently, their change in volume will be less than that of cells with fewer ancestors. This is true even if these cells actually made a good attempt at coping with the environmental pressure (self-made men) and usually, even if the aristocrat's sons didn't solve the problem at all. Dividing by the change in volume assuming no metabolic activity gives us a rough measure of how much work was done by the individual metabolism rather than the ‘family’ (ancestry's legacy).

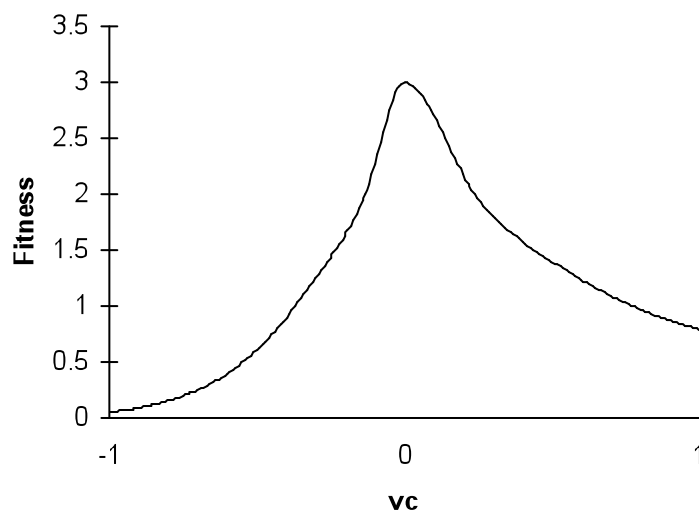


Figure 8.1: change in volume fitness component

Cell Lifetime Metric

The second fitness metric addresses the length of time the cell existed and is given by

$$u = \text{MIN}(T_{\text{cell}}, T_{\text{MAX}}), \text{ then} \quad (8-2)$$

```

if  $u < 1.0$ 
    return 0
else
    return  $\log_{10} u$ 

```

Finally, we normalise u by dividing it by $\log_{10} T_{\text{MAX}}$. The reason we take the minimum of the time the cell lived, and the maximum simulation time is the same as that given above.

Switch Correlation Metric

The next metric addresses the correlation between the chemical species used to modify the cell membrane and the promoter regions of the operons. We wish to evolve cells whose genes are switched by changes in these chemicals. That is, cells that can use information in the metabolism to respond to possibly dangerous or other changes in the environment.

At the end of the simulation of a cell, we calculate the correlation between chemicals in the cell able to change the cell membrane and their ability to regulate the inducible and repressible operons on the genome. For each chemical species in the cell that can be used as a builder or breaker, we calculate its capacity to regulate each of the operons and sum these values. This number is then weighted by the concentration of the chemical. The values for each applicable chemical in the cell are totalled and divided by the number of operons in the cell to allow comparison between cells. Typically this final value is in $[0, 0.1]$, so we multiply by 10 to scale back to $[0, 1]$.

Reaction Graph Complexity Metric

The fourth fitness metric measures the complexity of the metabolic reaction graph using a function of the number of reactions possible in the cell. Here we use a simple exponential function to provide an asymptote to 1. This occurs when there are approximately sixty reactions available in the cell.

$$rc = 1 - e^{-0.1r} \quad (8-3)$$

where r is the number of reactions available to the cell. Not all of these reactions may be active in the cell, however. Only reactions that have the available reactants and enzymes in sufficient quantities will be active.

Membrane Modification Metric

The next fitness metric relates to the number of species of membrane building and breaking chemicals available to the cell. Again we use a simple exponential function of the number of membrane modifying chemicals in the cell (at the end of simulation) to determine a fitness component in $[0, 1]$.

$$bc = 1 - e^{-0.5s} \quad (8-4)$$

where s is the number of membrane building or breaking chemical species in the cell. The constant multiplier 0.5 gives a fitness component of almost one when s is about ten. All cells are initially provided with one building chemical (mostly *o44*) and one breaking chemical (often *o30*) yielding a fitness component of around 0.5.

Parsimony Metric

The final fitness component is a parsimony constraint and as such, it is in direct conflict with some of the other constraints. It acts to keep the complexity of the cell low. In a pragmatic sense, this stops our cell simulations from getting so large and slow that they

can't effectively run. We use the number of differential equations in the cells as a measure of complexity.

$$pc = e^{-5.0 \times 10^{-11} d^4} \quad (8-5)$$

where d is the number of differential equations in the cell. The constant multiplier 5.0×10^{-11} gives a fitness component of about 0.5 when there are 210 differential equations and around 0 when there are about 800.

The six fitness metric values are multiplied and summed in each possible combination. Finally, we divide the fitness value by 2 if the cell died. Cell death occurs when any chemical concentration exceeds 1.0×10^{-4} .

Selection of Parents

Roulette wheel selection is used to identify parents for breeding. Sometimes we employ a strategy where all fitness values in the breeding population are decremented by the lowest fitness in the population. This operation, which tends to decrease genetic diversity, is applied before roulette wheel selection to give an additional disadvantage to low scoring individuals. None of the experiments in this thesis apply this operation.

Population Management and Replacement Strategies

In section 2.5, we reviewed two methods of managing the population of genomes: two generations with discrete generations and Rosenberg's single population with blended generations. Our implementation favours the latter method.

Typically, there are more simulations in a population than there are free processors. Not all the simulations, therefore, can be run in parallel, so to maximise processor usage, we do not employ discrete generations of cells. Parents, children and ascendants remain in the population together. In our study, the size of population ranges from 50 to 200 but remains static for the duration of the particular experiment. Most simulations utilise a

population of 100 individuals. Breeding only occurs when a significant fraction (0.75) of the population have finished their simulation and therefore have a fitness score.

No chemical reactions occur in the parents while breeding takes place. All activity in a cell halts when its simulation finishes.

Simulations take different amounts of time to run depending on the complexity of their differential equations and the fitness of the cell. Often cells that die do so after a small number of time steps. Cells reaching a predefined time step are stopped. Typically after they reach time 1.5×10^5 . A numerical integration algorithm with adaptive step size control is used for the simulation of each cell. Simulations that exceed a predefined number of time steps are also terminated: typically after 2500 steps.

Replacement Strategy Used

We employ three strategies to choose the cell in the population to be replaced with the new child cell.

Strategy 1: Replace Lowest Scoring Cell

We use an elitist (Goldberg 1989) strategy where we replace the lowest scoring cell. It is allowable for a cell to remain in the population for the entire duration of the genetic algorithm. This scheme is represented in figure 8.2 by the line with average fitness going highest. Looking at this line, replacing the weakest cells appears to be a sensible scheme: population convergence is the best of the three schemes. The primary problem with this scheme, however, is that cells that perform well but always bear inferior children can pollute the population and gene pool. Such a cell arises if it earns a high fitness, but leaves its chemical concentrations in such a fragile state that a child cannot keep the chemicals in a useable state. Often such parents would fail soon after, themselves. Even breeding such a parent with itself will produce weak children. This high line shows a population of high-scoring, but inferior, parents. Towards the end, most of the population consists of these inferior cells. The unfit children that result continue to be swept from the population. They show up as little dips on the average fitness line.

Strategy 2: Replace Oldest Cell

We replace the cell that has been in the breeding population the longest time regardless of its fitness. This scheme removes the poor parents, but allows underperforming cells to remain in the population so that those cells with high fitness are eventually lost.

When a run is characterized by a few highly fit cells that often produce poor children, these fit cells, which may be the best hope for future fit cells, are quickly swept out of the population by the poor children. The loss happens quickly because the simulations of low scoring cells occur faster than high scoring cells. High scoring cells are simulated to a larger time value. Many stillborn but rapidly evaluated children can sweep a highly fit parent out of the population before the simulations of fit children terminate and can breed with the fit parent. This problem can also be solved by using a more discriminating fitness function and by running the simulations to a higher time value. The latter technique ensures that only the more stable cells are retained. This strategy is represented in figure 8.2 by the average fitness line that goes lowest.

Replacing the oldest cell, with no regard for its fitness, gives a lower average fitness because weak cells are allowed to proliferate and breed. The maximum fitness is found later than with the other strategies because fit parents are lost. Short term solutions, that produce weak children (see above), flush out reasonably fit parents. That's why the line goes low.

Strategy 3: Alternate Between 1 and 2

A strategy that seeks the middle ground between the other schemes is to alternate between them. On even numbered breedings the lowest scoring cell is replaced and on odd numbered breedings we replace the oldest breeding cell. The middle line in figure 8.2 represents this strategy. This scheme gives us the advantages of both schemes and ameliorates the disadvantages by effectively doubling the amount of time a cell stays in the breeding population. Typically, we employ this strategy.

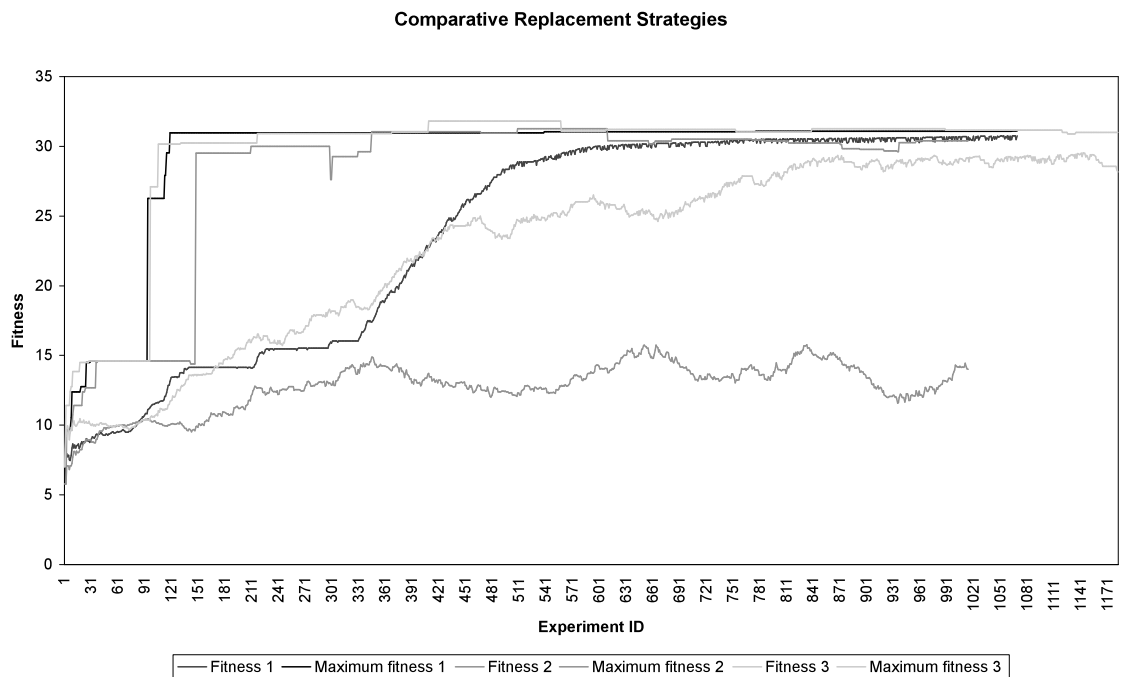


Figure 8.2 Comparative Replacement Strategies

8.2 Effects of Running Over a Network of Processors

The evolution of the cell simulations is done using two computer programs. The first program, called *gacell*, is the genetic algorithm shell. It is solely concerned with the population of genotypes, reproduction and genetic operators. For each genotype, it creates a new process to simulate the cell. This second program, called *cellsim*, builds the phenotype, runs the simulation and calculates the fitness. Each *cellsim* program is executed on a different processor. The *gacell* program, then, spawns the *cellsim*s over a network of processors. Usually around 15 or 20 SPARC machines are used for each run of the genetic algorithm.

Cellsim is a very large computer program that uses much memory. The processors that execute *cellsim* are not powerful machines so they must be solely dedicated to running the simulation. In addition, during semesters the machines must be shared with other students. This means that the cell simulations only run overnight and on weekends. At 7:00 am on weekdays, the operating system gives *gacell* a suspend signal to stop it

receiving CPU time and terminates all running cellsim programs. At 10:00 p.m., gacell wakes up, realises that its cell simulations aborted and continues its evolution, resubmitting the terminated cellsims.

The nocturnal existence of the cells, network interactions, and the fact that simulations run for differing amounts of time introduces a further random element into the genetic algorithm.

8.3 Program Verification

Much effort was expended verifying the two computer programs. The computer programs were written using the object-oriented language C++ which requires writing a number of classes (see appendix D).

A strategy of white-box testing was employed by writing a test program (harness) for every class. This tested the consistency of the class and the operation of every line of code in the program.

In addition, verification of important algorithms in the program was performed separately as follows.

The numerical integration algorithm was tested on a simpler system of differential equations with a known solution. This simpler system of equations used was the Lotka-Volterra equations (Hofbauer and Sigmund 1988). These model populations engaging in predator-prey behaviour.

Differential equations produced from a large number of genomes and initial conditions were individually verified. We checked the values of differential equations calculated by the program with hand-calculated values.

Operation of the genetic algorithm was verified on a simpler function with known maximum fitness. We also checked the genetic operators by viewing bit strings before and after the operation.

Chapter 9

BOOTSTRAPPING PROBLEM

Which came first: the chicken or the egg?

- Unknown

In this chapter, we present the results of the evolution experiments of populations of cells to cope with two different environments. Each environment exerts pressure on the cell. Is it possible to evolve cells that can exist in one of these environments from random beginnings? It turns out that it is indeed possible to evolve cells to perform this task.

Chapters 3 to 8 describe the cell model and the simulation program used to evolve populations of cells. Here we discuss the major experiments performed on the cells. In the next chapter we recount other interesting experiments made to the cells.

Initially, in this chapter we describe the bootstrapping problem. Next, we introduce the two environments in which evolutionary experiments occur and describe the pressure each exerts on cells. Following this we give results for experiments in these environments and show how cells evolved to cope with the environmental pressure. Finally, we make a number of comments about the cell solutions, the relationship between cells and their parents, and the Lamarckian effect of the passing of the mother's initial concentrations.

9.1 The Bootstrapping Problem

As we stated in section 3.1, each cell consists of two parts: a metabolism and a genome. Each part has a different evolutionary path. The metabolism and genome are tightly coupled. In well-adapted cells, the metabolism and genome work together. Many configurations of genome and initial chemical concentrations, however, will produce cells with abnormalities. For example, the metabolism may poison the cell with a particular chemical, or the cell may not be able to respond to an important environmental stimulus. The bootstrapping problem involves finding metabolisms (ie, from initial chemical concentrations) *and* genomes that work synergistically to produce cells that are stable in a given environment. This problem apparently cannot be solved solely by the genome. The context (initial chemical concentrations) for the genome is important. We use the fitness function for the genetic algorithm to determine whether the cell is stable. We will see in section 9.5 that this function is not completely able to differentiate between stable and unstable cells.

9.2 Environments for the Cells

Two simple environments are devised for the cells.

Environment A promotes cell shrinkage by bathing the cell with chemicals that breakdown the cell membrane. If the cell does not act against this pressure, it will decrease in size. A decrease in size causes an increase in concentrations for all chemicals in the cell because the number of water molecules in the cell lessens. This increase in concentrations causes two effects in the cell. There is an increase in the speed of reactions. The other effect is that chemicals that already had high concentrations now become closer to poisoning the cell. Recall that when a concentration becomes greater than 1.0×10^{-4} , we assume that the chemical is lethal to the cell. If the cell cannot respond to the environmental pressure, it can quickly move towards death, gathering speed as it continues to shrink in a positive feedback death spiral. The longer the cell waits before acting, or takes to act, the more difficult it is to stop the shrinkage.

Environment B has the opposite effect. It causes cell growth by supplying a higher concentration of membrane building chemicals than in the cell. When membrane building chemicals diffuse into the cell the number of water molecules in the cell increases (because the cell is larger) and the concentrations of chemicals in the cell decrease. This causes reactions to occur more slowly. At the same time, the surface area of the cell increases and this causes diffusion to increase. If the cell doesn't act on the influx of membrane building chemicals in time, it can soon find itself unable to arrest the increasing rate of diffusion because reactions (including enzyme production) have slowed to a snail's pace.

Table 9.1 presents the chemical species and their concentrations in each environment. The initial concentrations of completely random cells are also given. These cells form the initial population.

Both environments contain a number of other chemical species held at similar concentrations to the cell. This is done to focus the environmental pressure on cell size changes rather than on other tasks. As well, we give the environment and cell some simple chemicals to use as building blocks and all the monomers in high concentration to allow gene expression. Cells are also given a chemical species suitable for gene expression (a 'spider') in high concentration.

Table 9.1: Chemical and concentrations in environments A, B and parentless cells

Species	Env A	Env B	Cell	
<i>o0</i>	5.0×10^{-5}	5.0×10^{-5}	5.0×10^{-5}	Monomers for polymer construction
<i>o1</i>	5.0×10^{-5}	5.0×10^{-5}	5.0×10^{-5}	
<i>o2</i>	5.0×10^{-5}	5.0×10^{-5}	5.0×10^{-5}	
<i>o3</i>	5.0×10^{-5}	5.0×10^{-5}	5.0×10^{-5}	
<i>o4</i>	5.0×10^{-5}	5.0×10^{-5}	5.0×10^{-5}	
<i>o5</i>	5.0×10^{-5}	5.0×10^{-5}	5.0×10^{-5}	
<i>o6</i>	5.0×10^{-5}	5.0×10^{-5}	5.0×10^{-5}	
<i>o7</i>	5.0×10^{-5}	5.0×10^{-5}	5.0×10^{-5}	
<i>o8</i>	5.0×10^{-5}	5.0×10^{-5}	5.0×10^{-5}	
<i>o9</i>	5.0×10^{-5}	5.0×10^{-5}	5.0×10^{-5}	
<i>o01</i>			5.0×10^{-7}	Building blocks for spiders
<i>o23</i>			5.0×10^{-7}	
<i>o30</i>	5.75×10^{-7}	5.0×10^{-7}	5.0×10^{-7}	Membrane breaking species
<i>o44</i>	5.0×10^{-7}	5.75×10^{-7}	5.0×10^{-7}	Membrane building species
<i>o0123</i>	5.0×10^{-5}	5.0×10^{-5}	5.0×10^{-5}	Spider
<i>o5432</i>	1.0×10^{-6}	1.0×10^{-6}		Building block species
<i>o2468</i>	1.0×10^{-6}	1.0×10^{-6}		
<i>o9673</i>	1.0×10^{-6}	1.0×10^{-6}		
<i>o7345</i>	1.0×10^{-6}	1.0×10^{-6}		

9.3 Environment A Experiment

The environment A experiment uses initial chemical concentrations for the environment and initial cells as given in Table 9.1. All parameters have been given in chapters 3 through 8 as the typical values. We used a population size of 100 individuals. Since we

have a breeding threshold of 0.75, the size of the breeding population is 75. The other 25 individuals are either running or waiting for a processor to become available so that they can run. An important point to be made is that facilitated diffusion was not used for any of the experiments in this chapter. Genomes can specify carrier molecules and the metabolism can build them, but the carriers cannot bind to permeant molecules. Therefore, facilitated diffusion is turned off. We did this because facilitated diffusion caused the simulations to run too slowly. Later we ran experiments allowing facilitated diffusion.

Figure 9.1 shows a graph of the fitness attained during the run. The X-axis represents each experiment run. Values near the maximum fitness were reached very early. This occurs soon after children of the initial population start registering (after 75 experiments have run). The average fitness is calculated over all finished experiments in the population, excluding those that aborted. As the run progresses, individuals are removed from the population (in the manner described in section 8.1) to make way for offspring. The average fitness is calculated only on members currently in the population. The individual fitness of each experiment is also given. Experiments that were aborted register zero fitness. Since the fitness function halves all cells that died, values on the lower half of the graph mostly represent cells that died. The graph shows that most of the population becomes fit.

There are two kinds of cell in the population: those that were living at the end of their simulation and those that died. In all the experiments we conducted there is always a proportion of cells in the population that died during their simulation. Average fitness includes both classes of cell. It is, therefore, a misleading indicator of trends of either class in isolation. However, it does show trends of the entire population.

In all of the experiments in this and the next chapter, there is a large diversity of genes. Environments do not have one “right” solution with maximum fitness. There is a variety of valid approaches for solution of the environment and all have similar (high) fitness. This allows the population to maintain different genes that solve the problem of an environment as long as valid contexts for their use (ie, initial metabolic conditions) are available.

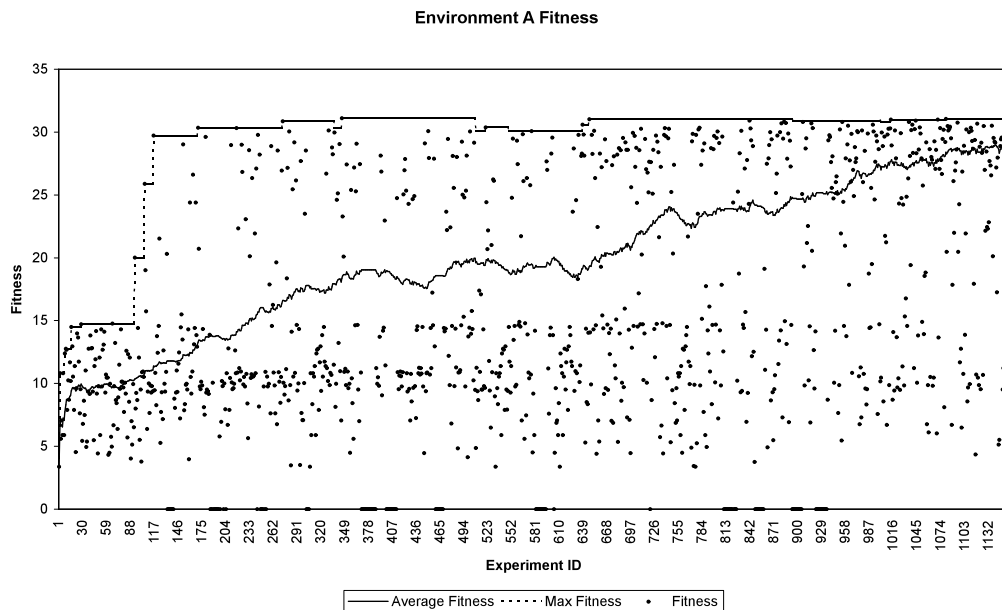


Figure 9.1 Fitness for environment A

Figures 9.2 and 9.3 plot the fitness of each individual against the fitness of each parent. Members of the initial population do not have parents, so they are plotted against zero. Points above the line $y = x$ on the graph represent offspring that were less fit than their respective parents and points below represent offspring that were fitter than their parents. There appear to be four main clusters in both graphs. The bottom left cluster represent offspring that died from parents that had died. The top left cluster shows children that died from living parents. Bottom right shows children that lived from parents that had died and the top right cluster shows children that lived from living parents.

There appear to be more points above the line $y = x$ than below. This indicates that more offspring perform *worse* than their parents.

The bottom left and top right clusters in figure 9.2 (the mother) are tighter and more closely aligned with the line $y = x$, than in figure 9.3. Indeed, in figure 9.3, the points in the top right cluster appear to be spread uniformly. This implies that offspring tend to have fitness similar to their mother rather than their father. The difference between parents is that initial chemical species are inherited only from the mother cell, whilst the

genome is inherited from both parents. The context of the genome, then, has an important effect on a cell's fitness.

Correlation coefficients given in table 9.2 clearly show that offspring fitness is more closely correlated to mothers than to fathers. Living offspring from living parents are correlated. Since the fitness of dead cells is halved, we estimate fitnesses less than 16 or 18 to be for cells that died.

Table 9.2 Correlation coefficients for environment A

Both fitness values are ...	Correlation coefficient between offspring fitness and mother's fitness	Correlation coefficient between offspring fitness and father's fitness
... over 16	0.284483	0.112108
... over 18	0.205816	0.146869

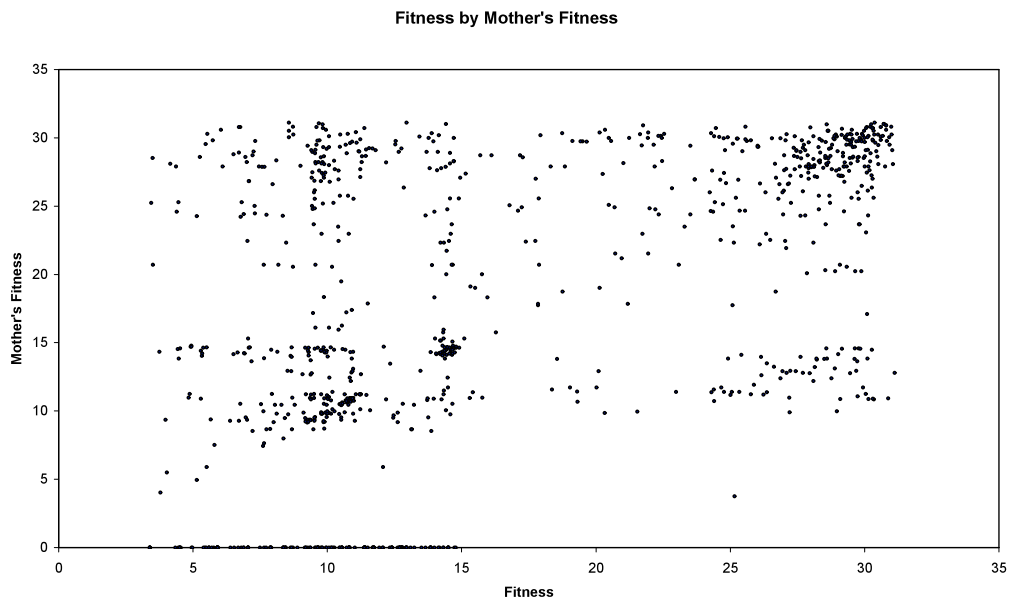


Figure 9.2 Fitness by Mother's Fitness

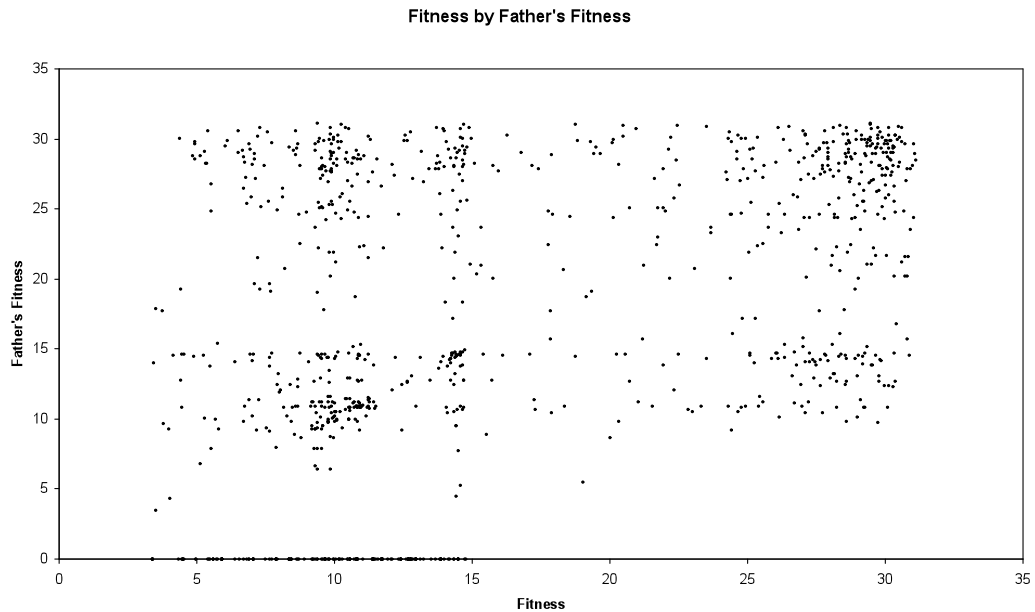


Figure 9.3 Fitness by Father's Fitness

Taking the final population of cells, we continued running each cell for a further time of 1.5×10^5 and calculated another fitness value. From this, we plotted the difference between the two fitnesses (new-old) against the old fitness. This graph is shown in figure 9.4. Points above the X axis show cells whose strategy could be sustained for the further time period. Points below mark cells in the final population that employed shortsighted solutions. Points close to the X axis represent stable cells. Many of the individuals in the final population implement short term solutions.

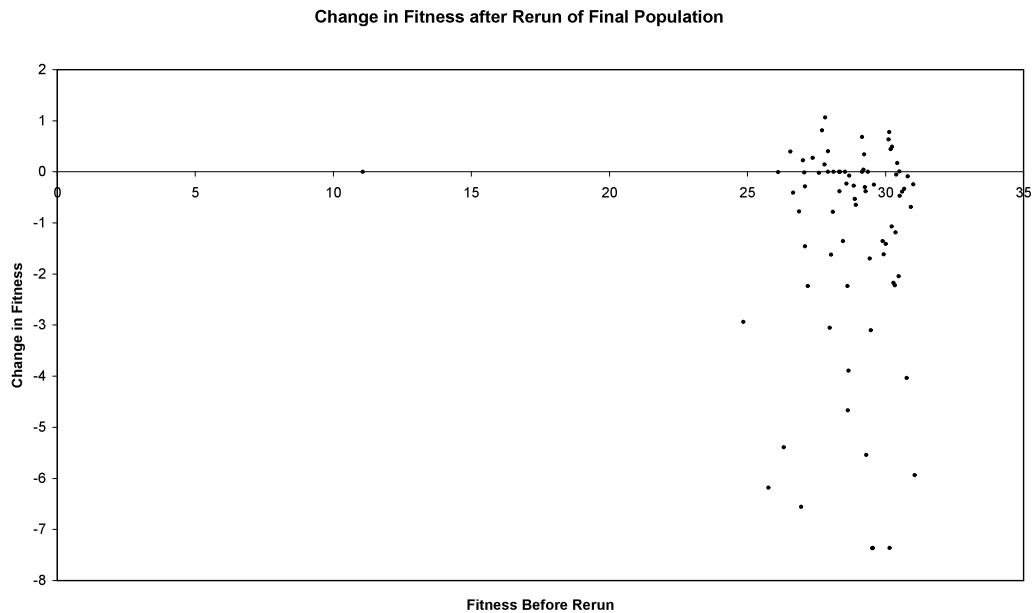


Figure 9.4 Change in Fitness after Rerun of Final Population

Inspection of these cells revealed that they exploited short-term solutions to the environment. One manifestation of a solution exhibiting such myopia is to bind one or more undesirable chemical species with another species. In the short term, the concentration of unwanted species is less because it exists as a bound complex with an enzyme. This gives rise to a higher fitness. Over a longer period, however, the short-term solution exhibits a problem. The undesirable chemical eventually comes out of its bound complex, which is saturated. The product of the reaction often has analogous chemical properties to the original unwanted species because it has a similar shape. However, it is now longer and therefore diffuses out of the cell slower. The shortsighted solution, in this case, makes the problem worse and offspring inherit chemicals yielding a lower fitness.

The main problem faced is that the fitness function is not able to differentiate between these myopic parents and those that produce offspring as fit as themselves. When scoring the parent, the fitness function must either examine the fitness of offspring or search for short-term solutions in the parent's metabolism. The former task is impossible because a cell must be scored before it issues offspring. The latter task is

difficult because it presupposes solutions to the environment and therefore particular kinds of cells. One of our goals with the design of the fitness function was to keep it as abstract as possible. As far as possible, the fitness should be determined from the actions of the cell rather than details of the actual enzymes coded or the reaction graph embodied in the metabolic pathways.

The problem of myopic parents doesn't appear to occur in Nature. Perhaps this is because organisms exist in very large populations, they live for a long time compared to our cells, and cellular evolution has been in existence for a very long time.

Primitive cells are thought to have arisen 3.5 to 4 billion (10^9) years ago and eukaryote cells 1.5 billion years ago. (Alberts, Bray et al. 1994)

The duration of the cell cycle varies greatly between cell types: from 8 minutes (for a fly embryo) to more than one year (for mammalian liver cells). A fairly rapidly dividing mammalian cell divides every 20 minutes (Alberts, Bray et al. 1994). Prokaryotes from some species divide every 20 minutes (Solomon, et al. 1993).

Later in this chapter, we will describe experiments aiming to reduce the number of myopic cells.

Let us examine two cells from this environment in more detail.

Case Studies of Individual Cells in Environment A

The first cell we will dissect is the cell in the final population that scored the highest fitness (31.0556). This cell contains six operons (See table 9.3).

Table 9.3 Operons in first example cell.

Operon	Switching	Genes	% switched on
0	Constitutive	<i>c18</i>	100
1	Inducible {522}	<i>c076</i>	28
2	Constitutive	<i>c05871, c677</i>	100
3	Constitutive	<i>e59</i>	100
4	Inducible {124}	<i>e53</i>	44
5	Constitutive	<i>c1</i>	100

Most of the operons in the cell are not useful because they code for carrier molecules. Carriers aren't used in these cells. Genes coding for carriers appear at random or are selected to boost the switch correlation fitness component.

Table 9.4 Membrane building and breaking molecules in the first example cell.

Building chemicals	Concentration (at beginning of run)
<i>o44</i>	5.4×10^{-7}
<i>o744</i>	2.45×10^{-15}
Breaking chemicals	
<i>o30</i>	6.2×10^{-7}
<i>o302</i>	2.5×10^{-15}
<i>o530</i>	None
<i>o630</i>	2.5×10^{-15}
<i>o830</i>	1.3×10^{-8}
<i>o5302</i>	None
<i>o8302</i>	2.5×10^{-15}
<i>o734530</i>	None
<i>o7345302</i>	None

This cell had only one spider molecule (*o0123*), which it gets by default from the initial conditions. The membrane building and breaking molecules available are listed in table 9.4. Only two enzymes exist in the cell, those being *e53* and *e59*, the enzymes listed on the genome. The enzyme *e53* is used to build many of the breaking chemicals (*o530*, *o5302*, *o734530* and *o7345302*). The other building and breaking chemicals were inherited from the mother.

This cell line has lived for a long time because the concentrations of *o30* and *o44* are higher than in the environment. This can only occur if the cell has shrunk considerably. As the total of the membrane breaking concentrations is greater than the building concentrations the cell will continue to shrink.

The strategy used by this cell seems strange because the genome instructs to build membrane breaking molecules – but it is reducing the amount of *o30* by putting it in bound states. This is a similar strategy to its ancestors. See *o302* for example. In the short term, this is a good strategy, but it breaks down in the long term. Initially, this cell achieved a fitness of 31.0556. When it was run longer, until time 300 000, the fitness for this second 150 000 time steps was only 25.1239.

Unfortunately, the cell has lost the ability to build *o744*. Presumably, this is because that ability did not translate to an increase in fitness. Production of *o744* temporarily puts *o44* and *o744* into bound state and therefore decreases the fitness of the cell. There will always be some amount of *o44* lost to the bound complex. In the long term, however, the fitness will recover. This sensible strategy, however, can't compete against the flashy short-term solution already mentioned.

The second cell we will examine is the cell in the final population whose fitness improved by the most when it was run for the further time period. Initially, the cell scored 27.8078. It improved by 1.0634 to achieve 28.8712.

Table 9.5 Operons in second example cell.

Operon	Switching	Genes	% switched on
0	Inducible {9338}	<i>c9</i>	42
1	Inducible {283}	<i>c25, c78, c677</i>	39
2	Inducible {3}	<i>e770</i>	17
3	Constitutive	<i>e53, e59</i>	100
4	Constitutive	<i>c3804468, c3367</i>	100
5	Constitutive	<i>c4</i>	100

This cell, as shown in table 9.5, contains six operons. There is a relationship between some of the genes in this cell and those of the first cell described. Operon 1 in this cell codes the carrier *c677*. This carrier also appears in operon 2 of the former cell. The carriers before *c677* in both cells share values. Operons 3 and 4 from the previous cell are combined into one operon (number 3) in this cell. Again, many of the operons aren't useful as they code carrier molecules.

Additional spider molecules are available to this cell. They are *o0123* (3×10^{-5}), *o01238* (2×10^{-8}), *o70123* (none) and *o701238* (none). The last two of these are constructed using the enzyme *e770*. The strategy used by this cell does not differ too much from the first cell. It uses enzymes *e53* and *e59* to bind more of the existing membrane catalysing species into complexes so that they cannot shrink the cell. As well, the cell has already shrunk so much that the concentrations of *o30* and *o44* are higher than in the environment and, thus, will diffuse out. The cell achieved a high fitness when the run was continued, although it uses a short-term strategy. This is because it can bind *o30* and *o07345* into many different complexes simultaneously.

Table 9.6 Membrane building and breaking molecules in the second example cell.

Building chemicals	Concentration (at beginning of run)
<i>o44</i>	5.28×10^{-7}
Breaking chemicals	
<i>o30</i>	6×10^{-7}
<i>o530</i>	None
<i>o07345</i>	2.4×10^{-15}
<i>o073453</i>	None
<i>o073459</i>	None
<i>o707345</i>	None
<i>o734530</i>	None
<i>o0734530</i>	None
<i>o073459673</i>	None
<i>o2468734530</i>	None

9.4 Environment B Experiment

The experiment for environment B has exactly the same parameters as experiment A. The only difference is in the chemical concentrations in the environment. We use the values in the third column of Table 9.1. Environment B forces the cell to grow larger.

Figure 9.5 shows the fitness of each experiment, the average fitness of the population at each time, and the maximum fitness. As with environment A relatively fit individuals evolved quickly. Figure 9.6 plots the graphs in figure 9.1 and 9.5 together. The average fitness for both environments starts out similarly and at the end both populations are mostly filled with fit individuals. Fitness of cells in environment B, however, is consistently higher than those cells in environment A. As the membrane building and breaking coefficients are set to the same values, the simplest explanation for this is in the formulation of the fitness function. Recall from figure 8.1 that the volume component of fitness is a piecewise curve. Fitness for cells that grow a little is higher than for cells that shrink.

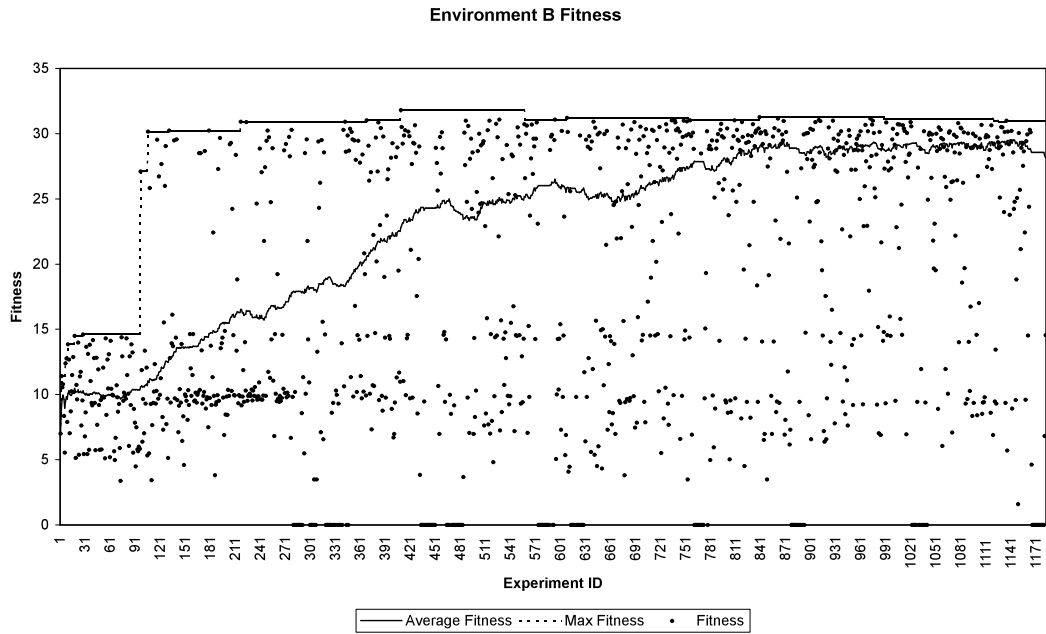


Figure 9.5 Fitness for Environment B

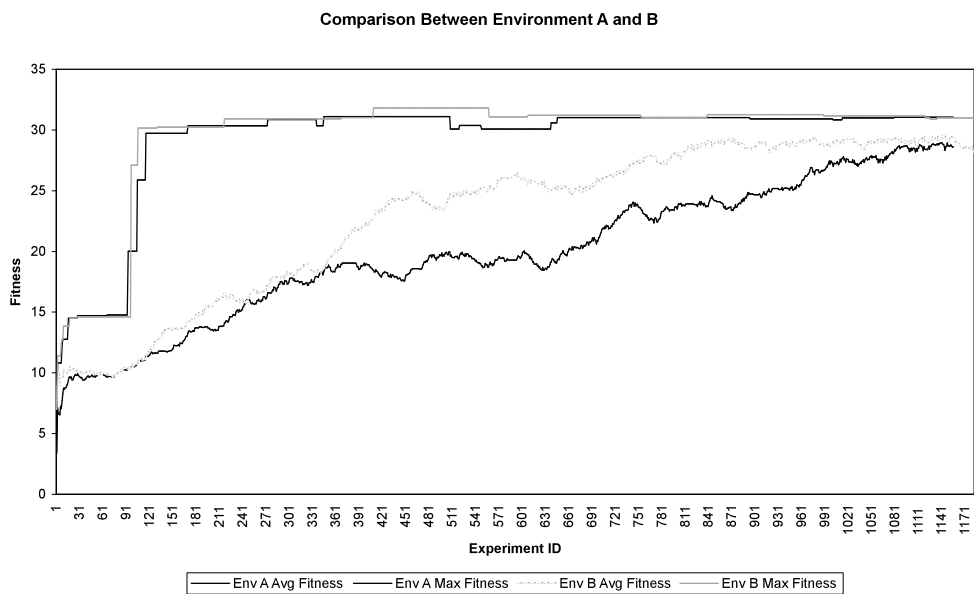


Figure 9.6 Comparison between Environments

Figures 9.7 and 9.8 show the fitness compared to each parent's fitness. Results are very similar to environment A. Again, fitness is more correlated with the mother's fitness than with the father's. See table 9.7.

Table 9.7 Correlation coefficients for environment B

Both fitness values are ...	Correlation coefficient between offspring fitness and mother's fitness	Correlation coefficient between offspring fitness and father's fitness
... over 16	0.460134	0.050393
... over 18	0.495231	0.015153

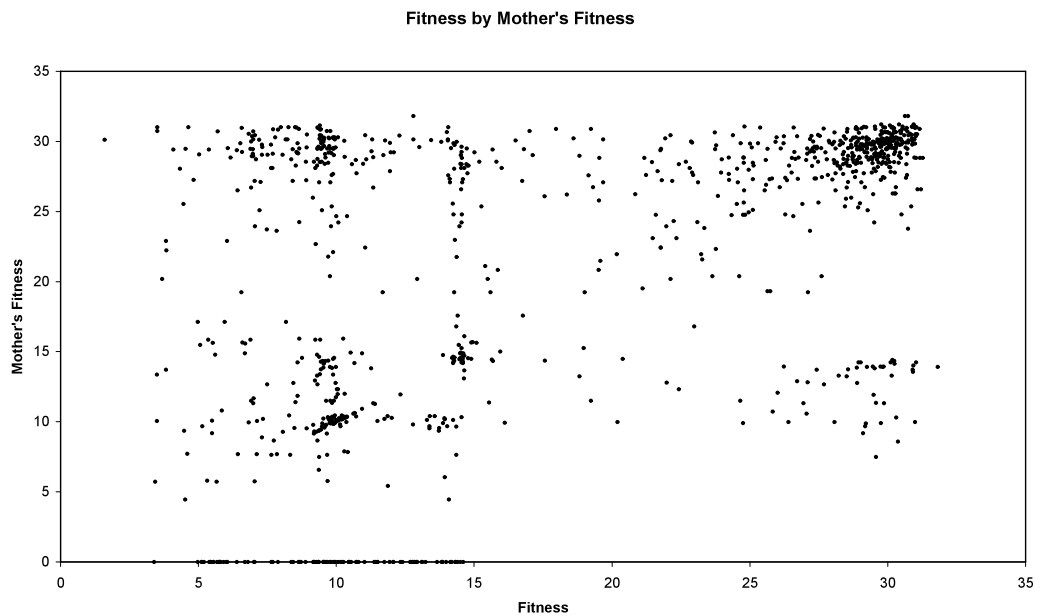


Figure 9.7 Fitness by Mother's Fitness

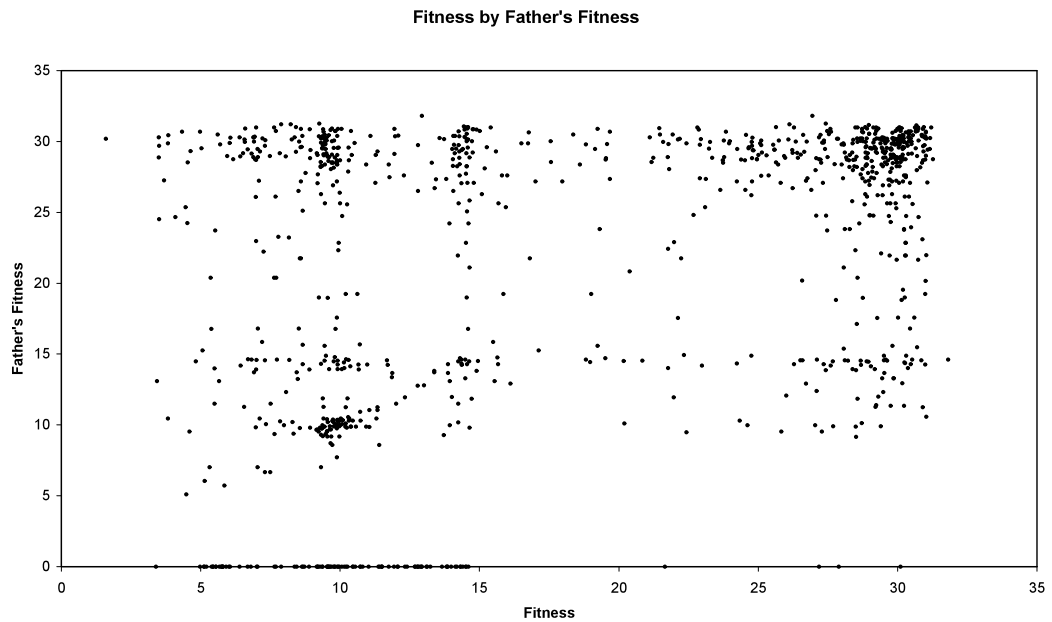


Figure 9.8 Fitness by Father's Fitness

As with figure 9.4, figure 9.9 graphs the change in fitness observed when the final population of cells in environment B are run for a further time period of 1.5×10^5 . A feature of this plot that wasn't as obvious in figure 9.4 is that the group of points leans slightly to the left. That is, lower fitnesses tend to be more stable than higher fitnesses. The final population consists of two kinds of cells. Myopic cells implementing short-term solutions tend to have higher fitness than their more stable cousins.

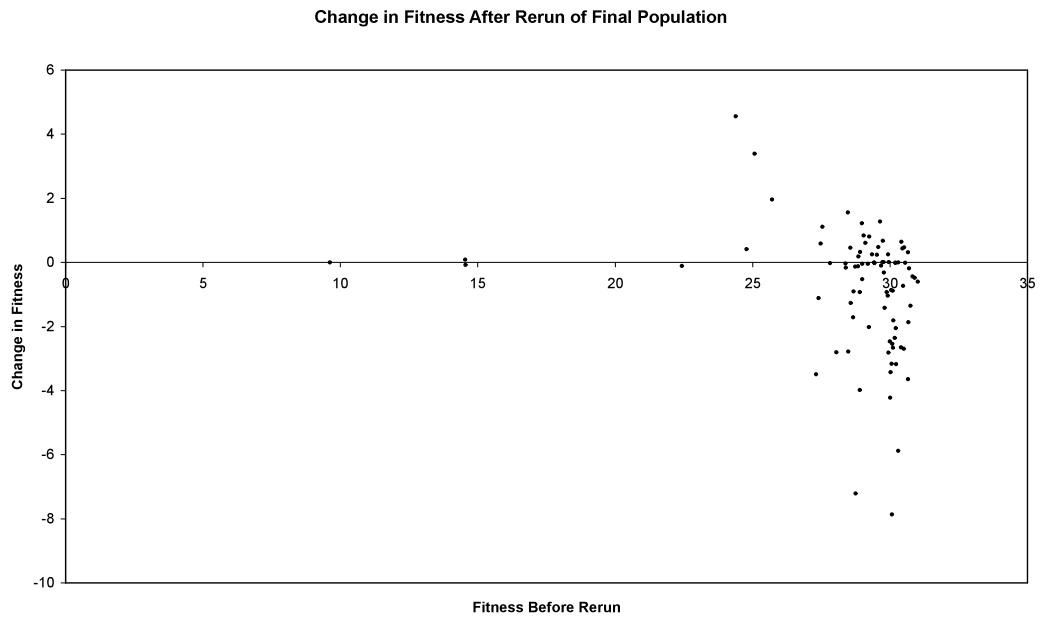


Figure 9.9 Change in Fitness After Rerun of Final Population

Case Studies of Individual Cells in Environment B

We will now examine two cells living in environment B.

The first cell we will look at scored the maximum fitness in the final population. It scored 31.0092 initially and 30.4144 for the second time period. It contains the 5 operons shown below in table 9.8.

Table 9.8 Operons in third example cell.

Operon	Switching	Genes	% switched on
0	Constitutive	<i>e51709</i>	100
1	Inducible {351}	<i>c0</i>	40
2	Constitutive	<i>c57</i>	100
3	Inducible {066}	<i>c02640, e137</i>	21
4	Repressible {1648}	<i>c00</i>	46

Three spider species are available for transcription in the cell. These are *o0123* (2.8×10^{-5}), *o01237* (7.54×10^{-14}) and *o10123* (7.54×10^{-14}). Table 9.9 lists the rich variety of membrane building and breaking species that the cell can use.

Table 9.9 Membrane building and breaking molecules in the third example cell.

Building chemicals	Concentration (at beginning of run)
<i>o44</i>	5.43×10^{-7}
<i>o446</i>	1×10^{-15}
<i>o447</i>	1×10^{-15}
<i>o73451</i>	None
<i>o7345130</i>	None
Breaking chemicals	
<i>o30</i>	4.73×10^{-7}
<i>o130</i>	9.8×10^{-16}
<i>o1309, o5130, o09673, o13093, o13099, o25130, o096737, o096738, o1309673, o7345130, o13096737, o13096738, o54325130</i>	None

The enzyme *e51709* is used extensively in this cell. The main strategy in the cell is to build membrane breaking molecules to fight against the *o44* molecules that are diffusing in. This appears to be a sound strategy since the rerun value is similar to the original fitness. There are, however, three weaknesses with the strategy. The same enzyme (*e51709*) that is used to build many of the membrane breaking species also makes builder molecules. The enzyme is not sufficiently specific. As well, the builder molecules that are constructed are long, and therefore, will diffuse out very slowly. The next flaw in the strategy is the molecule *o7345130* that is made. This molecule acts as both a builder and breaker molecule. As such, its production is a waste of valuable resources. The final weakness is that many of the breaker molecules are produced using the existing breaker species *o30* and *o130*. These chemicals become bound and will reduce the overall fitness of the cell because they are not available to catabolise the membrane. This, though, will increase their rate of diffusion into the cell!

The other cell in this environment that we will examine was the cell in the final population that recorded the largest increase in fitness when it was run for the additional time period. Initially it scored 24.3847. After continuing the run, its fitness rose 4.5622 to 28.9469. Table 9.10 shows its nine operons.

Table 9.10 Operons in fourth example cell.

Operon	Switching	Genes	% switched on
0	Constitutive	<i>e07</i>	100
1	Inducible {0}	<i>e619</i>	15
2	Inducible {45}	<i>c369009</i>	20
3	Repressible {6}	<i>e53</i>	93
4	Repressible {9}	<i>c62892071</i>	35
5	Inducible {549}	<i>c2</i>	94
6	Constitutive	<i>c001</i>	100
7	Constitutive	<i>c09</i>	100
8	Inducible {751718}	<i>e38</i>	55

As well as the four enzymes encoded on its genome, this cell has inherited another enzyme from its mother (*e06*). This chemical will soon degrade into its two constituent molecules. The cell has five spiders at its disposal: o0123 (2.46×10^{-5}), o01238 (none), o96730123 (6.1×10^{-8}), o196730123 (none) and o967301238 (none). Table 9.11 shows how it can modify the membrane. The cell has managed to build only membrane breaking chemicals, many of which don't rely solely on modifying the important chemical *o30*.

Table 9.11 Membrane building and breaking molecules in the fourth example cell.

Building chemicals	Concentration (at beginning of run)
<i>o44</i>	5.43×10^{-7}
Breaking chemicals	
<i>o30</i>	4.7×10^{-7}
<i>o96730123</i>	6.1×10^{-8}
<i>o07, o075, o306, o307, o530, o0735, o0755, o0757, o3075, o5530, o7530, o7345, o07575, o30735, o30755, o30757, o57530, o73530, o75530, o073455, o073457, o303345, o307575, o734530, o757530, o0734575, o0755432, o0757345, o3073455, o3073457, o5734530, o7345530, o30734575, o30755432, o30757345, o73457530, o75734530, o073455432, o073457345, o196730123, o967301238, o3073455432, o3073457345, o7345734530</i>	None

9.5 Discussion

What techniques did the cells discover to cope with the environments?

The cells discovered a number of different techniques to master their environments, some of which were described above. Some of these strategies were shortsighted and led to problems for later generations. Some appear counterintuitive.

The most straightforward action a cell can take is to produce more of a chemical needed to balance the equilibrium of the building and breaking chemicals. This usually means a

chemical species from the ‘opposite’ family to chemicals diffusing in with the highest rate.

Operons coding enzymes that control the manufacture of these chemicals can be induced by the presence of the opposing chemical, or repressed by the presence of the same chemical. However, this didn’t always manifest. Switching appears to be a refinement on the production process and, in these environments, when the operon needed to be expressing continually, didn’t necessarily yield a higher fitness than a constitutive operon. Perhaps environments that vary over time may evolve switching more easily. We tended not to see enzymes that affected both builders and breakers.

Cells can produce these opposing chemicals in two ways. Existing membrane modifying chemicals can be changed. Chemicals that do not affect the membrane can also be combined to build the opposing chemical. The latter scheme is preferable because the existing opposing chemicals are not put into a bound state whilst the new chemicals are built.

When cells add chemicals to the opposing chemicals, these chemicals temporarily move into a bound state. Bound opposing chemicals cannot be used to fight the environment. The products of these reactions, however, have a longer shape, and therefore diffuse out of the cell much slower. This technique is useful when there is a higher concentration of the opposing chemicals inside the cell than in the environment.

A counterintuitive approach that some cells take is to produce more of the chemical that is diffusing in by joining it to another chemical in the cell. This causes the chemical that is flushing in to become bound and therefore ineffectual to the cell membrane. This is an example of where the cell makes a choice that looks good in the short term, but is disastrous in the long term. As soon as the product chemical appears in the cell, it will be much more difficult to get rid of because it is now longer and therefore diffuses out more slowly.

Another behaviour we observe takes a number of generations to accomplish. Cells that produce more spider molecules can produce more enzymes faster and therefore solve the problem more quickly. Similarly, over a number of generations, cells eventually

produce more enzymes, for progressively better solutions. The knowledge of how to produce these chemicals (that is, the operon with the enzyme that catalysed the reaction that produced the chemical) is often lost. This happens when some of the chemical is passed to children but not the operon with the enzyme that made the chemical. As long as the resultant chemicals are long, and no reactions evolve to destroy them, they remain in the cell.

Short genes for enzymes appearing close to the start of operons are better than longer genes or genes that appear late in the operon because they are expressed earlier and faster. The lag time between switching the operon and seeing the effect is also less for these genes.

Weeding out the Shortsighted Solutions

A novel method to rid us of the shortsighted parents has been devised. This involves modifying the fitness of an individual once the fitness of its issue becomes known. If a child cell is fitter than a parent is, the parent's fitness is increased. Conversely, if a child cell is less fit than a parent is, the fitness of the parent is decreased. The latter alteration compensates for the myopic parents in the population, lessening their probability of future parentage. Modifications can be made to either or both parents.

Such a scheme is non-biological. There is, however, a stage where the simulation becomes divorced from biology. In our simulation, this stage occurs at the level of the population. The sexual reproduction and retroactive change of parent's fitness schemes used in this simulation are an attempt to search a state space rather than accurately model biological behaviour. The retroactive change of fitness described in this section is also required to deal with the Lamarckian effect exhibited in the simulation. If it was computationally feasible to run simulations with very large populations or for long periods of (simulated) time, the need for retroactive change of parent fitness would be reduced or eliminated.

A simple algorithm is used to calculate the fitness to be applied to parent. Initially, all cells in the population are ordered according to their score. Next, we find the fitness of

the cell midway between the parent and child cells. If there are an even number of cells between the child and parent, the fitness will be the average of the two cells closest to the centre. The parent's fitness is changed to this value.

Three experiments were conducted to gauge the efficacy of this algorithm. We ran one experiment changing the fitness of the mother cell retroactively based on the fitness of its offspring. This was compared to a run changing only the fitness of the father, a run changing the fitness of both parents and a control experiment. The environment B experiment, described above, was used as the control experiment, where no fitness values were changed. The other three experiments had the same parameters as the control experiment except for modification of parent fitness values. Average fitness and maximum fitness for each of these four experiments are graphed in figure 9.10. The final population from each experiment was run for a further time period to assess whether the cells had short or long term solutions. These results are plotted in figures 9.11 through to 9.14.

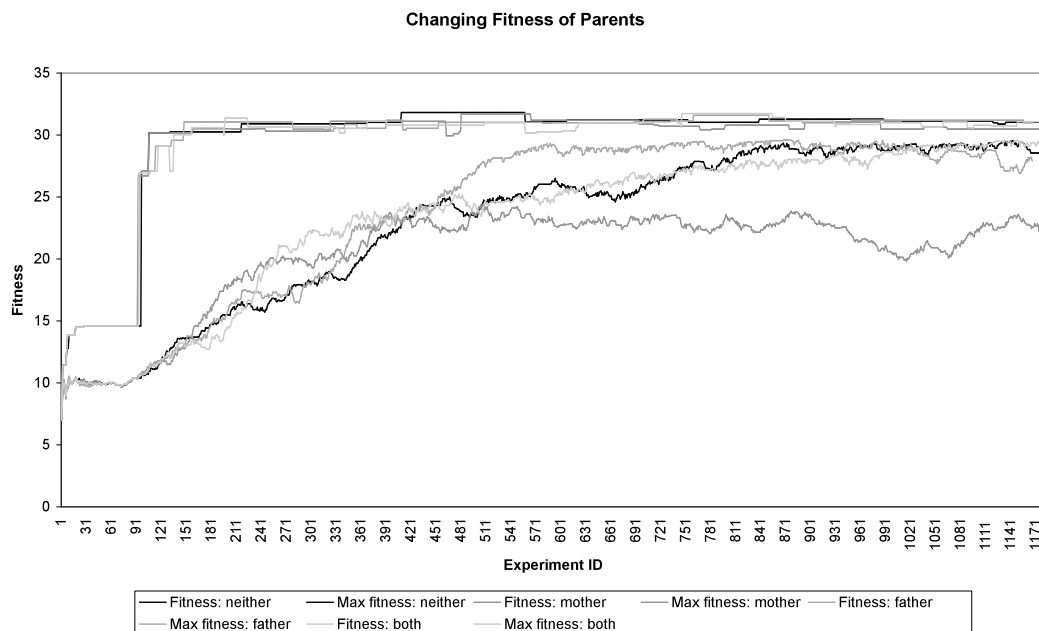


Figure 9.10 Change Fitness of Parents

The main points of notice in figure 9.10 are:

- Not changing the fitness of either parent – the average fitness runs a middle course but ends high
- Changing only the mother's fitness – the average fitness is the lowest of the four experiments and ends much lower
- Changing only the father's fitness – the average fitness climbs faster than the others and ends high
- Changing the fitness of both parents – the average fitness appears to be similar to the control experiment
- The maximum fitness of all four experiments are similar.

It is necessary to recall two important facts about the experiments to interpret these results:

- Fitness is more correlated to the mother than the father (see table 9.7 and figures 9.7 and 9.8);
- Cells exhibiting short term solutions tend to score higher on average than those with long term solutions but yield children that tend to die with a higher probability.

The maximum fitness lines of each experiment are similar. This, however, only shows that short term solutions are always present in the population to some extent.

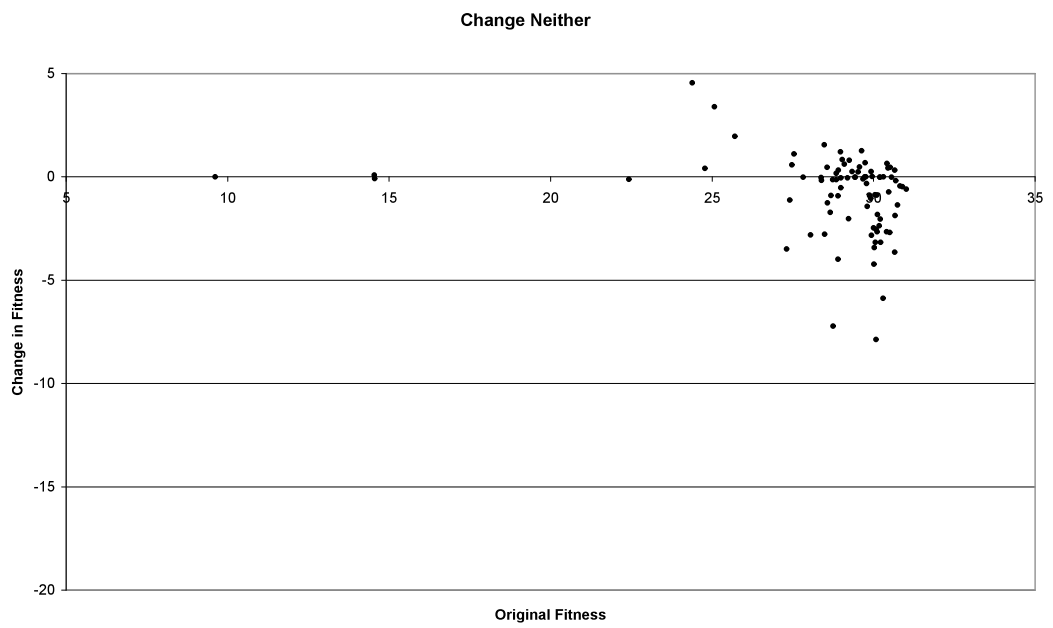


Figure 9.11 Rerun change in fitness when changing neither parent

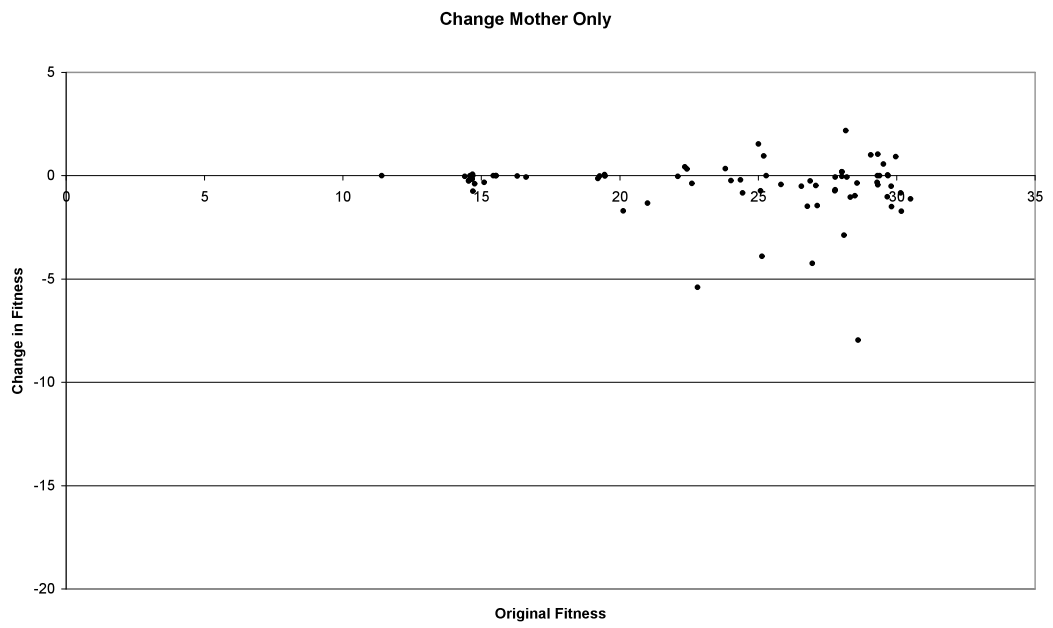


Figure 9.12 Rerun change in fitness when changing mother only

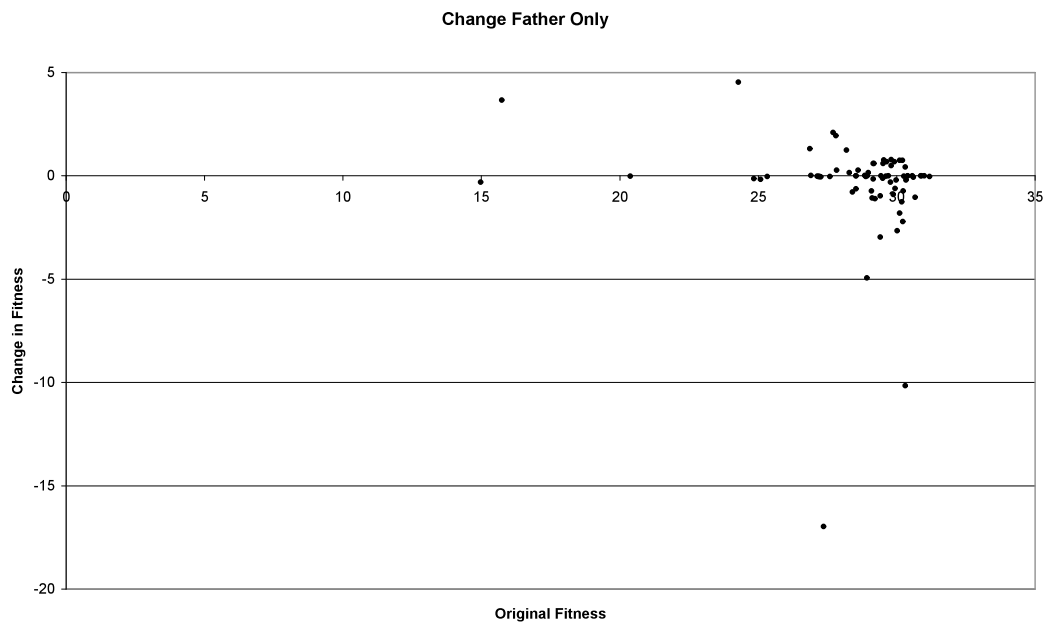


Figure 9.13 Rerun change in fitness when changing father only



Figure 9.14 Rerun change in fitness when changing both parents

For each experiment, table 9.12 lists the average fitness and standard deviations of cells in the final population (before the rerun was done), the average change of fitness after the extra running time (and standard deviation), and the estimated number of dead cells in the final population (before the rerun). The averages and standard deviations were calculated only for living cells. Cells are deemed dead if their fitness is less than 18.

Table 9.12 Average values and standard deviations for the fitness of living cells in the final populations and their rerun values and the estimated number of dead cells in each population.

Experiment	Average old fitness	SD old fitness	Average rerun delta	SD rerun delta	Number < 18
Neither	29.24209	1.519383	-0.85288	1.977294	3
Mother	26.43928	3.273452	-0.64026	1.625204	19
Father	28.87245	1.813762	-0.47446	2.589543	2
Both	29.6607	0.698706	-1.47775	2.680919	0

Changing the fitness of only the mother

Figure 9.12 shows the stability of solutions in the final population of cells when the fitness of a mother cell is retroactively modified once the fitness of her offspring is known. It shows that the final population exhibited a wider range of fitness values than in the other three experiments (see also the standard deviation in the third column of table 9.12). On average, cells in this population had lower fitness than the other experiments (see table 9.12, column 2). Many of the cells in this population, however, had very low fitness values, reminiscent of cells that died when they ran. The standard deviation of the changes in fitness after the extra running time is low (compared to the other experiments). This implies that there was stability in the cells' solutions and is characteristic of long term solutions. Lower fitness values are also a sign of long term solutions. Changing the mother's fitness contributes to the stability of solutions.

Why does this happen? Figure 9.12 gives a snapshot of the population at one moment: the end of the run. We need look at how the population changes over time or rather, how the sub-populations of cells with short and long term solution change over time. Table 9.13 shows the behaviour of each sub-population.

Table 9.13 Changes in sub-populations during the “Change Mother Only” experiment.

Mother	Father	Characteristics of Child	How parents fitness is changed
Short term	Short term	Short term with high fitness or death	Short term mother will stay high or drop considerably
			Short term father will remain high
Short term	Long term	Short term with high fitness or death	Short term mother will stay high or drop considerably
			Long term father will stay medium
Long term	Short term	Long term with medium fitness	Long term mother will stay medium
			Short term father will stay high
Long term	Long term	Long term with medium fitness	Long term mother will stay medium
			Long term father will stay medium

Tallying the effect on each component of the population, we find

- Short term individuals: two ways to stay high or drop considerably and two ways to remain high
- Long term individuals: likely to remain with medium fitness values

This effect needs to be iterated, taking selection into account, to see how the population changes. Long term individuals will tend to maintain a background medium to low fitness level. The short term cells act differently, though. Since their fitness values tend, on average, to be higher, they are selected for breeding more often than the long-term individuals. Some of the short-term individuals that mother offspring will have their fitness reduced considerably. This will occur when the metabolic values they pass on to their children move towards critical values. The number of short term mothers lessen because the ones whose fitness has decreased are less likely to breed again and are eventually selected out. Cells exhibiting short-term behaviour that fathered offspring remain to breed again, but when they act as mothers there is a chance that they will have their fitness decreased. The fate of the short term mothers acts as a sink. Eventually the cells with short-term solutions will be lost when their solutions start not to work any more.

Changing the fitness of only the father

Figure 9.13 shows the stability of solutions in the final population of cells when the fitness of a father cell is modified once the fitness of his offspring is known. Fitnesses end much higher than when changing the mother's fitness. Many of the rerun values lie close to the x-axis or are above. These are the stable solutions, or those that do better when run for longer. These tend to be at the lower fitness values. The higher fitness values tend to lose fitness when run for longer. A few cells lost large amounts of fitness when run for longer. In particular, one probably died. There were more viable cells in the population than in the "Change Mother Only" experiment.

Table 9.14 Changes in sub-populations during the “Change Father Only” experiment.

Mother	Father	Characteristics of Child	How parents fitness is changed
Short term	Short term	Short term with high fitness or death	Short term mother will remain high
			Short term father will stay high or drop considerably
Short term	Long term	Short term with high fitness or death	Short term mother will remain high
			Long term father will be set to high or drop considerably
Long term	Short term	Long term with medium fitness	Long term mother will stay medium
			Short term father will be set to medium
Long term	Long term	Long term with medium fitness	Long term mother will stay medium
			Long term father will be set to medium

Table 9.14 shows how parts of the population change over time. Tallying the effect on fitness of the short and long term individuals, we find

- Short term individuals: two ways to remain high, one way to remain high or drop considerably and one way to be set to medium
- Long term individuals: one way to be set high or drop considerably and three ways to remain medium

Compared to the “Change Mother Only” experiment, there are less ways for the short term solutions to drop out completely and there’s one way for the selection of a long term solution to be increased. This will tend to keep the average fitness higher than when changing only the mother.

Changing the fitness of both parents

The most striking aspect of figure 9.14, which shows the stability of solutions in the final population of cells when the fitness of both parents are modified, is that fitness values are more concentrated. As well, many cells are stable, but there is a definite ‘tail’ of cells that performed much worse when run for the extra time period. These were not present in the runs that changed one of the parent’s fitness.

Table 9.15 Changes in sub-populations during the “Change Both Parents” experiment.

Mother	Father	Characteristics of Child	How parents fitness is changed
Short term	Short term	Short term with high fitness or death	Short term mother will stay high or drop considerably
			Short term father will stay high or drop considerably
Short term	Long term	Short term with high fitness or death	Short term mother will stay high or drop considerably
			Long term father will be set to high or drop considerably
Long term	Short term	Long term with medium fitness	Long term mother will be set to medium
			Short term father will be set to medium
Long term	Long term	Long term with medium fitness	Long term mother will be set to medium
			Long term father will be set to medium

Table 9.15 shows how parts of the population change over time. Tallying the effect on fitness of the short and long term individuals, we find

- Short term individuals: three ways to remain high or drop considerably and one way to be set to medium
- Long term individuals: one way to be set high or drop considerably and three ways to be set to medium

As in the “Change Father Only” experiment, there is a way to increase the fitness of long term solutions. This raises the average fitness in the population. Short term solutions should be replaced quickly because both parents will be penalised when a poor child is produced. Unfortunately, long term solutions may be lost in this way, as well. There may be an undercurrent of short term solutions. Once there are a large number of long term solutions, short term solutions that father offspring to long term mother will have their fitness reduced slightly.

Effect of the Lamarckian Pressure

The interplay between the Lamarckian style evolution of the initial chemical conditions in the cell and the Darwinian evolution of the genome is important for the solution of complex environments. The Lamarckian evolution allows cell lines to progressively solve the problem.

Cell lines can use Lamarckian evolution to allow events that start very slowly to build up until they occur fast enough to be of benefit to descendants. We observed cell lines that eventually built up enzyme and spider concentrations to values large enough to allow reactions to occur quickly in the lifetime of one cell.

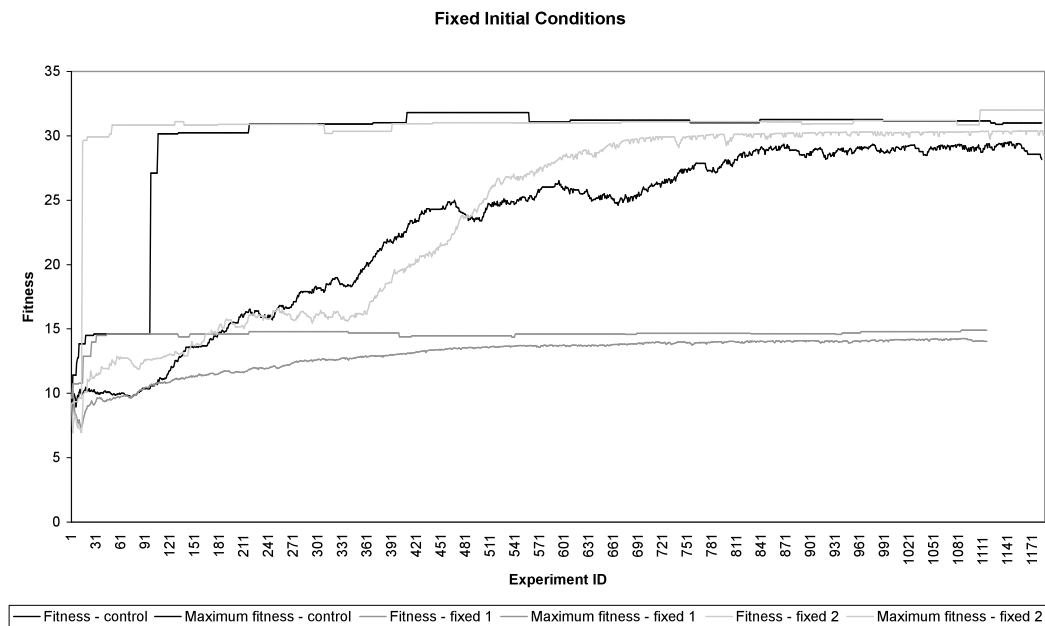


Figure 9.15 Fixed Initial Conditions

Three experiments are represented in figure 9.15:

- A control experiment (initial conditions for cell are not fixed and are passed to the child from the mother cell). This is the same experiment as in environment B, above.
- “Fixed 1” sets all initial conditions to the same as at the start of the control experiment. All offspring receive these same initial metabolic conditions. The graph shows that the cell could not solve the environment. All cells died. These initial concentrations may not afford a solution in the amount of time the cells live.
- “Fixed 2” sets the initial metabolic conditions the same as those of a cell from the control experiment’s final population. The cell chosen was one that lived and was stable. The change in fitness when this cell was run for an extra time period was 0. The experiment shows that the maximum fitness was found quickly but that convergence was slower than control. See section 10.4 for a discussion of this latter fact.

The Lamarckian effect of passing initial metabolic conditions to offspring makes it easier to find a solution to an environment.

Figures 9.16 and 9.17 plot the fitness of cells against the fitness of each parent for the second fixed initial condition experiment. Table 9.16 lists correlation coefficients similarly to table 9.7. Comparing table 9.16 with table 9.7 shows that the fitness of offspring with their mother's fitness for the experiment with fixed initial conditions is less than in the Lamarckian experiment. This reflects the fact that the mother does not bequeath her initial conditions in this experiment. Correlation between father and offspring is now closer to the mother (as expected). In this experiment, there is no difference between mothers and fathers with respect to their relationship with their children.

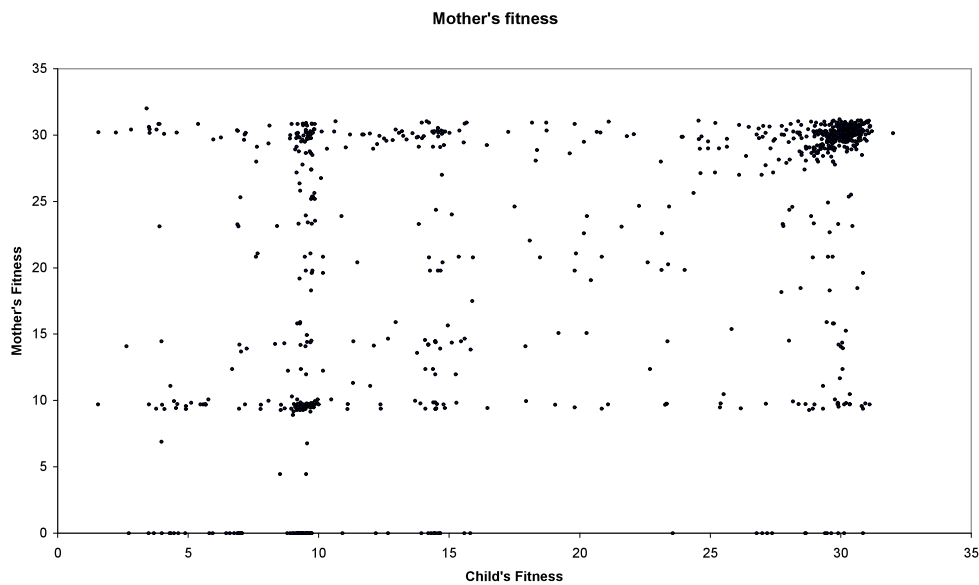


Figure 9.16 Child fitness by mother's fitness for fixed 2 experiment

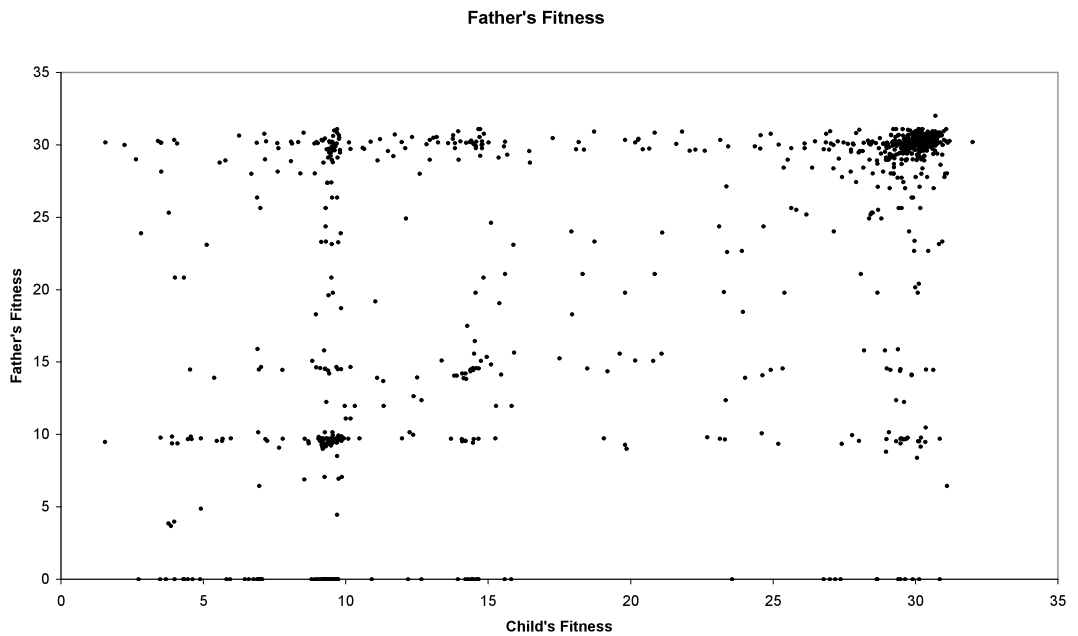


Figure 9.17 Child fitness by father’s fitness for fixed 2 experiment

Table 9.16 Correlation coefficients for fixed 2 experiment

Both fitness values are ...	Correlation coefficient between offspring fitness and mother’s fitness	Correlation coefficient between offspring fitness and father’s fitness
... over 16	0.426922	0.361765
... over 18	0.440323	0.342856

Chapter 10

OTHER EXPERIMENTS

Spirit constantly formulates questions. It must store up inquiries. The creative power of spirit lies in its ability to create questions. Thus the supreme objective of spirit is in the creation of the question itself – in short, the creation of nature. But that is impossible. Yet the march towards impossibility is the method of spirit.

- Yukio Mishima, “Forbidden Colours”

In this chapter, we give the results for a variety of other experiments performed on the cells. These experiments follow on from those described in chapter 9, where we were interested more in what solutions were found by the cells for two environments.

There are three motivations for the following experiments:

- sensitivity analysis
- effect of facilitated diffusion
- effect of variations on the genome (bits vs nibbles and single-strandedness vs double-strandedness).

Initially we wish to explore the sensitivity of evolution to the frequency of the three genetic operators: crossover, mutation and inversion. Next, we describe an experiment where we allow facilitated diffusion of permeants through the cell membrane. After this, we describe experiments comparing double-stranded bit genomes with double-stranded nibble genomes. That is, we compare genomes that permit crossover and inversion to occur between any bits with those that permit them to occur only on nibble boundaries. Finally, we compare the results of experiments performed with single-stranded genomes with those using double-stranded encodings.

Environment B is used for all of the experiments in this chapter. Comments about the average fitness and genes in the population made in section 9.3 are still valid because these experiments are still finding solutions to an environment although their focus is on some other factor. Reiterating from section 9.3, there is always a proportion of cells that die during their simulation and average fitness includes these cells. In addition, and importantly, genes in the populations are diverse because there is a variety of allowable solutions to the problem in the environment.

10.1 Genetic Operators

Experiments were performed that modified the rates of the genetic operators: crossover, mutation and inversion.

Crossover

Three experiments were run modifying only the crossover rate. The environment B experiment described in section 9.4 acted as the control experiment with a crossover rate of 0.04. Two other experiments were run: one with a crossover rate of 0.02 and the other with a rate of 0.08. Results for these experiments are graphed in figure 10.1. As described in section 5.7, the crossover rate applies to nibbles on the genome. A crossover rate of 0.04 (the control) gives four expected crossover points; 0.02 gives two and for 0.08, we would expect eight crossover points. Because these are all a fairly large number of crossover points, this explains why the populations perform worse as the crossover rate is increased.

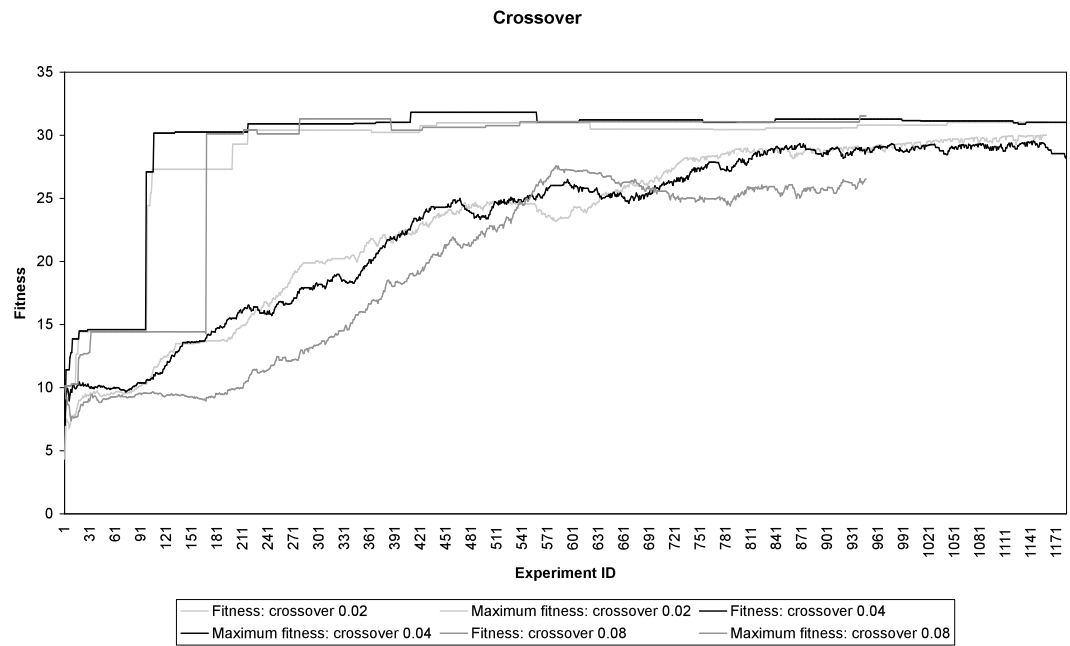


Figure 10.1 Vary Crossover Rates

Mutation

We conducted four experiments modifying only the mutation rate. As before, the environment B experiment described in section 9.4 acted as the control experiment with a mutation rate of 0.005. Three other experiments were run: with a mutation rate of 0.0005, 0.05 and 0.3. Results for these experiments are graphed in figure 10.2. As described in section 5.7, the mutation rate applies to bits.

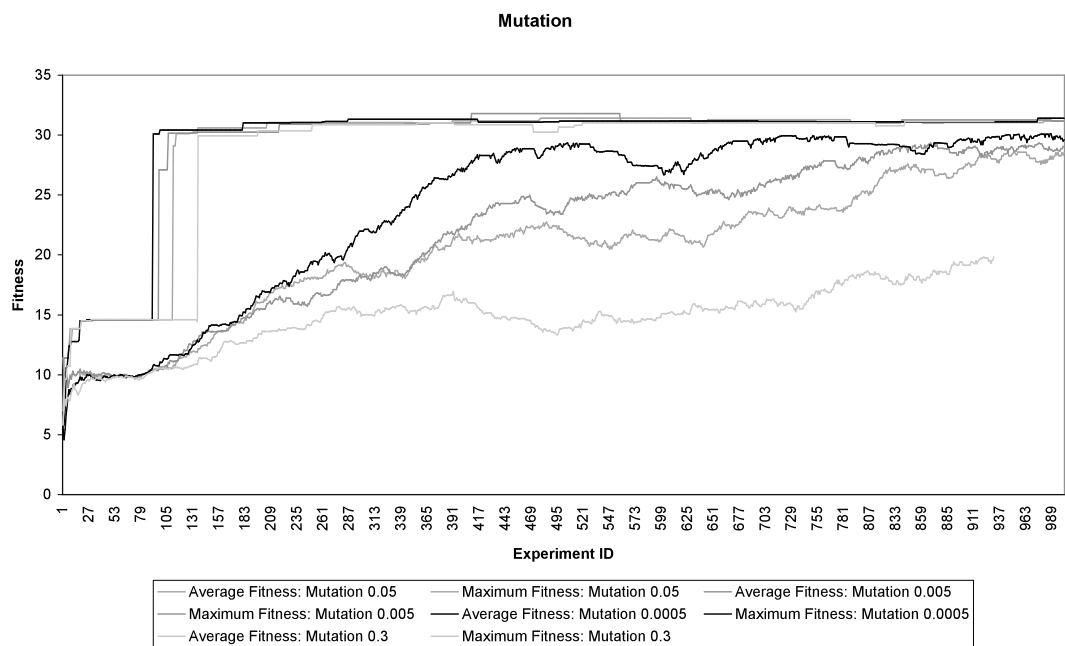


Figure 10.2 Vary Mutation Rates

Results suggest that the lower the mutation rate the faster the convergence of the population to the maximum fitness. In addition, initial discovery of cells with high fitness occurs later with higher mutation rates. Both of these observations are consistent with the fact that mutation adds diversity (or noise) into genes in the population. When the mutation rate is very high, the noise affects fit children adversely, slowing the rise of the average fitness.

We expect our genomic encoding scheme to be very sensitive to increased mutation rates because mutation of a base often affects many bases downstream from it. For example, changing a *<start operon>* base changes the meaning of all bases following it until the next *<start operon>* code. A similar problem exists for the *<start enzyme>* and *<start carrier>* bases. Apart from these simple experiments, we did not explore the detailed effect of changing mutation rates on the robustness of strings in the genomic language. This area warrants further study.

Inversion

Figure 10.3 graphs experiments that modified only the rate of inversion. The control experiment (environment B from section 9.4) uses an inversion rate of 0.15. We also conducted experiments with inversion rates of 0 and 0.3. As stated in section 5.7, the inversion rate is applied to the entire genome. In the control experiment, for example, there is a fifteen-percent chance that the inversion operator will be applied to the genome. The results demonstrate that an increased rate of inversion allows the population to converge towards the maximum fitness faster. This reflects the use of inversion in permuting the genome and building short order building blocks for crossover to manipulate. The inversion operator was not strictly necessary to solve the problem but it helped feed it into the population. Since the maximum fitness is found soon after breeding starts and is not much improved upon, we feel that the environment is not difficult for the cell to solve, or is difficult but can't be improved easily. The fact, then, that the inversion operator helps convergence, seems to be at odds with others' findings that inversion is only useful for highly nonlinear (difficult) problems (see section 2.4). This, in our experience, is not the case: convergence for our problem is ordinarily very slow, even when the environment is easy. Some reasons for this are given in section 10.4.

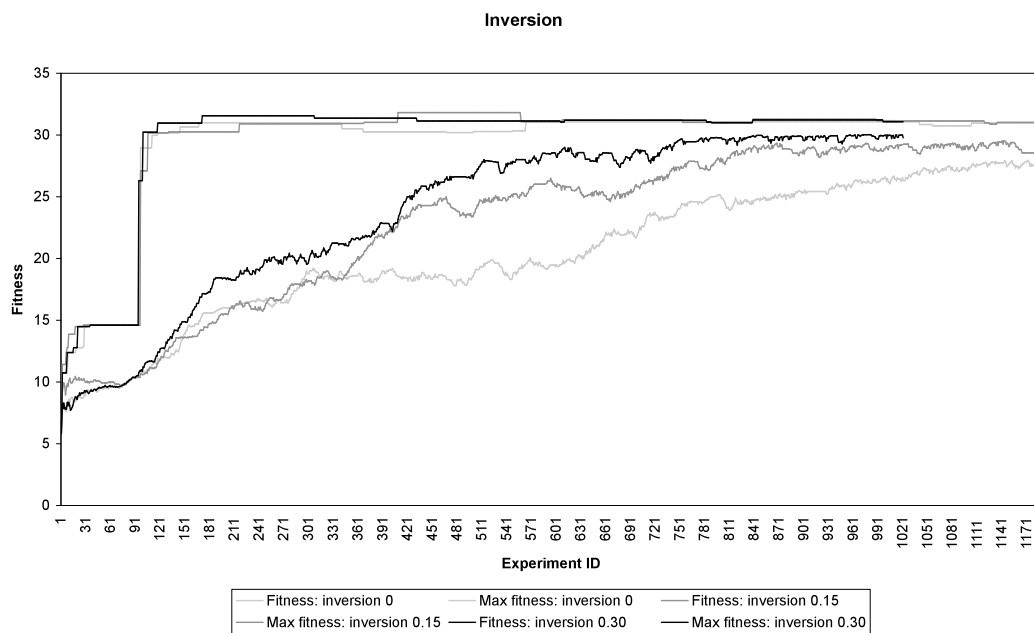


Figure 10.3 Vary Inversion Rates

10.2 Facilitated Diffusion

Figure 10.4 graphs experiments conducted examining the effect of facilitated diffusion. Two experiments were run: the control experiment did not feature facilitated diffusion; the other experiment allowed facilitated diffusion. The control experiment is the now familiar environment B experiment from section 9.4. Differences in parameters between the experiments are listed in table 10.1. Meanings of parameters are discussed in section 7.3.

Average fitness is quite similar for both experiments. The average fitness for the facilitated diffusion experiment is slightly lower than that of the control experiment for almost the entire run and maximum fitness values are mostly the same for this time. This suggests that facilitated diffusion allows more ways for a cell to decrease its fitness or more constraints for the GA to simultaneously optimise. Eventually a better average fitness was maintained for the facilitated diffusion. Maximum fitness for the facilitated diffusion experiment was initially much better than for the control experiment. This benefit was not passed successfully to offspring in this case because that cell did not breed. Other cells that were slightly less fit did breed.

Table 10.1 Difference in values for parameters between control experiment and facilitated diffusion experiment

Parameter	Control	Facilitated Diffusion
D_b (facilitated diffusion coefficient)	0	1.0×10^8
k_+ (carrier binding coefficient)	0	1.0×10^{-5}
k_- (carrier release coefficient)	0	1.0×10^{-1}

On examining cells from the final population, we found that fit cells did not use facilitated diffusion to handle their environment. Instead, facilitated diffusion was a hindrance. In the fittest cells, the carriers generated by the genome were almost always very short (eg, *c5*, *c2*). Our matching rules for the binding of carriers to permeants does not allow such short carriers to work. We were seeking longer, more specific carrier molecules. Some longer species of carrier molecule were generated by the genome, but in all of the cells we examined, these carriers were only able to bind to chemicals having very low concentration. Sometimes these potential permeants were membrane building or breaking species, but only those with very low concentration.

Cells found it difficult to utilise carrier molecules. There appear to be two reasons for this.

Firstly, carriers proceed much more slowly than enzymes. Three reactions must occur for a carrier to have done its job. The permeant must be bound, moved and unbound. Enzymes, on the other hand, only require one reaction: the one that they catalyse. Both proteins require the long spider-expression pathway. Perhaps the length of time required to reveal their benefit in the cell is comparable to the cell's lifetime or even greater. Investigation of one cell supports this view. At the start of its life, this cell had 62443 molecules of *c48* on the interior side of its membrane and one molecule on the outside. Permeants were available for transport to the outside of the cell. At the end of the

simulation, there were 66215 *c48* molecules inside the cell and only 1 on the outside. The cell had produced more carrier molecules, and perhaps bound some permeant, but no carrier-permeant complexes had unbound on the outer side of the membrane.

The second problem relates to how carriers match permeants. A carrier doesn't just match a single permeant molecule; it matches a (possibly large) family of similarly shaped molecules. To be more specific, a carrier must be longer and more completely like the permeant's shape. Longer carriers take more time to produce by the spider-expression pathway than shorter carriers. There tend to be large families of similar molecules in our cells, because our enzymes seem to join similar molecules in more complex ways. As a simple example, with the basic chemicals *o03* and *o3* and the enzyme *e33*, an entire family of chemicals (*o033*, *o0333*, *o03333* ...) are produced. A carrier such as *c053* will match against any chemical starting with *0* and having a *3* in the third position. That is, almost all of the family of chemicals generated by the enzyme. Typically, the families aren't as simple as this, but with two or more enzymes, large similar families are often generated.

Facilitated diffusion warrants further research. In particular, it would be interesting to increase the binding, diffusion and unbinding rates to try to speed up the effect so that evolution could use it. As well, it may be interesting to change the carrier matching rules.

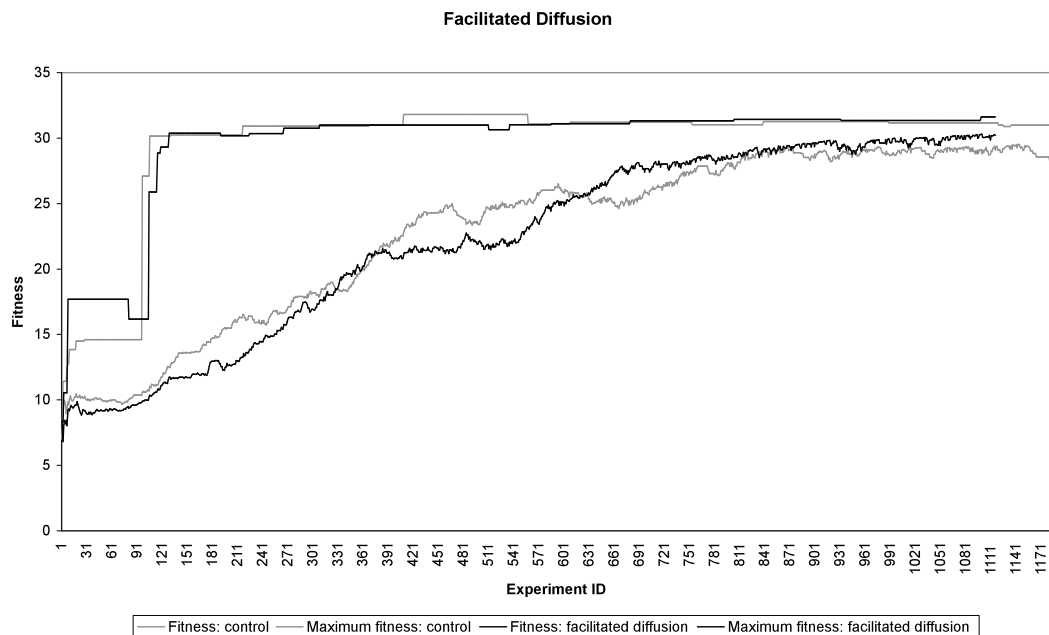


Figure 10.4 Facilitated Diffusion

10.3 Comparing a Bit Genome with a Nibble Genome

We are interested in whether a genome consisting of bits performed differently than our nibble genome. That is, whether the blocking factor was important. To this end, we ran an experiment where crossover and inversion were not limited to nibble boundaries. The genome still consists of two strands. Figure 10.5 compares an experiment using double-stranded bit genomes with the control experiment of section 9.4. The control experiment uses a double-stranded nibble genome. By this, we mean that breakages of the genome due to crossover and inversion only occur on nibble boundaries.

Holland (1992) suggests that genetic algorithms will work better with genomes with many detectors having only a few values (ie, a bit string) than with genomes with fewer detectors each having many values (for example, our nibble genome). Holland's reasoning for this is that, for encodings with a similar cardinality of genome space, the encoding scheme expressing the most schema will search genome space the most efficiently. Holland (1992) states that “[the difference in number of schemata for

different encoding schemes] suggests that, for adaptive plans which can use the increased information flow (such as reproductive plans [ie, genetic algorithms]), many detectors deciding among few attributes are preferable to few detectors with a range of many attributes”.

For a double-strand genome with l detectors and a attributes available at each detector, there are a^n possible single strands. Since the second strand is completely determined from the first strand, there are also a^n distinct double stranded genomes.

When calculating the number of schemata in a double-strand encoding, the value at each detector position on the first strand can be any one of the a attribute values or the “don’t care” value (“*”). That is, one of $a+1$ values. Values on the other strand can either be the value determined from the first strand or the “don’t care” value. That is, one of two values. There are, therefore, $(a+1)^n 2^n$, or equivalently, $(2(a+1))^n$ values.

The number of schemata represented by one double-strand genome is calculated similarly. Positions on both strands can take the “don’t care” value or the actual attribute value that is on the strand. So $2^n 2^n$, or 2^{2n} schemata are represented by a double-strand genome.

Each base of our genome is a detector. Although each nibble has sixteen values, only fourteen of these are distinct. This means that each detector in the nibble genome has fourteen attributes. There are roughly the same numbers of distinct genomes in a binary encoding of thirty bits (2^{30}) as in a nibble encoding of eight nibbles (where only fourteen are distinguishable) (14^8). The comparative numbers of schemata for the binary and nibble double-strand encodings are listed in table 10.2 along with numbers for their single strand equivalents. It is clear that vastly more schemata are available for binary encodings than for nibble encodings.

Table 10.2 Comparative numbers of genomes and schemata for four different encoding schemes

Encoding Scheme	Number of genomes	Number of schemata	Number of schemata that each genome represents
Single strand binary	$2^{30} \cong 1 \times 10^9$	$3^{30} \cong 2.06 \times 10^{14}$	$2^{30} \cong 1 \times 10^9$
Single strand nibble	$14^8 \cong 1 \times 10^9$	$15^8 \cong 2.56 \times 10^9$	$2^8 = 2.56 \times 10^2$
Double strand binary	$2^{30} \cong 1 \times 10^9$	$6^{30} \cong 2.21 \times 10^{23}$	$2^{60} \cong 1.15 \times 10^{18}$
Double strand nibble	$14^8 \cong 1 \times 10^9$	$30^8 \cong 6.56 \times 10^{11}$	$2^{16} \cong 6.55 \times 10^4$

When designing our encoding though, we chose the nibble encoding over a bit encoding because we felt that allowing the crossover and inversion operators to work at a bit granularity would introduce too much noise into the algorithm. When these genetic operators are constrained to breaking the genome at nibble boundaries, the bases at the ends are maintained. If the genetic operators are unconstrained and can act at any bit position, bits in the bases at the end points or at the crossover point are changed into another base. Unconstrained crossover acts as a crossover coupled with a mutation of the base straddling the crossover point. A similar effect occurs with unconstrained inversion.

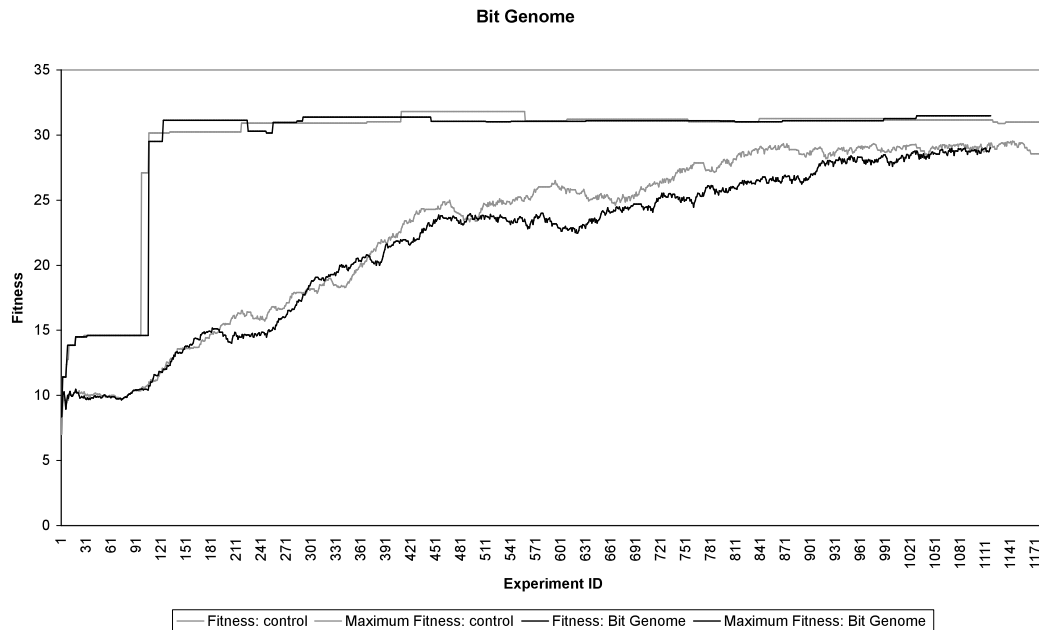


Figure 10.5 Double-stranded Bit Genome

Figure 10.5 shows that average fitness for the double-stranded bit genome is just less than that of the double-stranded nibble genome. The maximum fitness values are similar. This suggests that it is more difficult for the bit genome to produce fit children. Since the only difference between the two experiments is the addition of noise during crossover and inversion in the bit genome, this appears to be the likely candidate for the poor performance of the bit genome. It is also instructive to reflect on the fact that Holland used a 10-symbol alphabet for his broadcast language rather than encoding these symbols into binary blocks (Holland 1992). As we have stated before, this problem seems to be easy for the genetic algorithm. This is the reason why the benefit for the binary genome of encoding more schemata is not useful for environment solution. Perhaps for a more difficult problem, the importance of the differences in the number of schemata would outweigh the cost of additional noise during recombination and inversion.

10.4 Comparing a Single Strand Genome with a Double Strand Genome

We are interested in the benefit conferred on the cell by having a double-stranded genome instead of a single-stranded genome. Looking at table 10.2 we can see that the number of schemata associated with double-stranded genomes is far greater than for the single-stranded representations with no additional representational cost (in numbers of detectors). This extra information can be exploited by the genetic algorithm. In effect, the search becomes broader.

Compared to single-stranded encodings, though, the genetic operators have mysterious effects on schemata. Mutation, for example, simultaneously affects two bases: one on each strand. The effects of crossover and inversion are even more unusual.

Since the effect of the genetic operators on single-stranded encodings is different, it is difficult to create an experiment that fairly compares single-stranded encodings to double-stranded encodings. As a first approximation, we doubled the mutation and crossover rates for the single-stranded experiments. The inversion operator was not allowed for the double-stranded experiment, as we did not implement an analogue in the single-stranded coding. Our inversion operator relies on the topography of the double-stranded encoding. A single-stranded experiment was conducted with genomes the same length as one of the strands in the double-stranded experiment. A second experiment with single-stranded encodings used a genome with the same number of nibbles as in the double-stranded experiment. That is, double the number as in the first single-stranded experiment. The characteristic parameters for each experiment are listed in table 10.3.

Table 10.3 Parameters for the single-stranded and double-stranded genome experiments.

Parameter	Double-stranded	Single-stranded 1	Single-stranded 2
Genome length	100+100	100	200
Mutation rate	0.005	0.01	0.01
Crossover rate	0.04	0.08	0.08
Inversion rate	0	0	0

Figure 10.6 graphs the average fitness and maximum fitness for these experiments. Initially, the average fitness for the double-stranded experiment is higher and grows faster than either of the single-stranded experiments. Eventually, however, the average fitness values of the single-stranded experiments outstrip the double-stranded genomes. The increase in average fitness of the second single-stranded experiment (the one with genome of length 200) is much slower than the first single-stranded experiment.

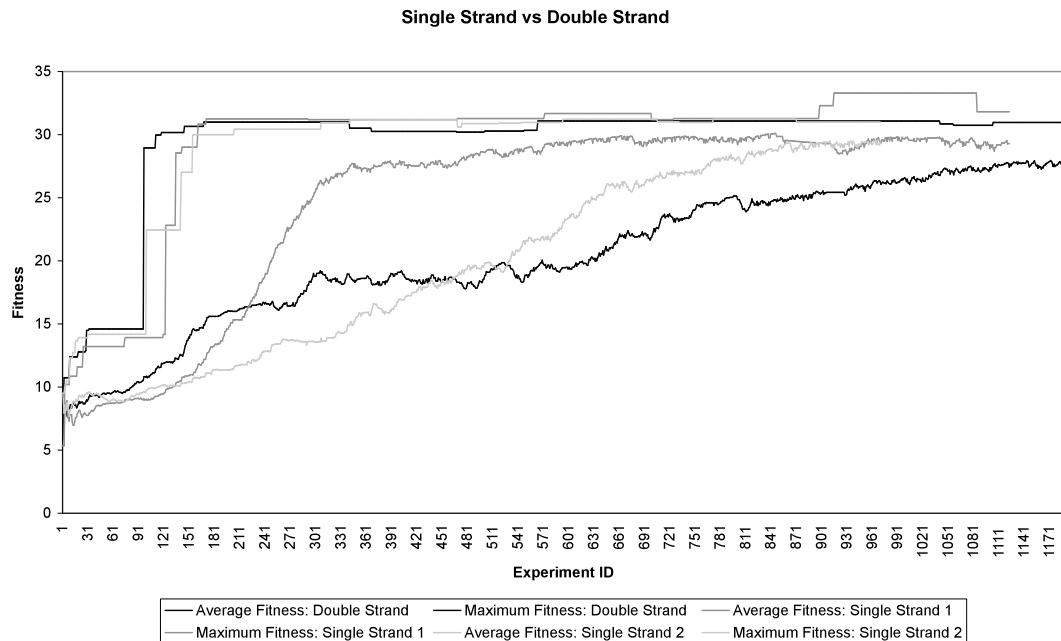


Figure 10.6 Single Strand Genome vs Double Strand Genome

The initial fast increase in average fitness of the double-stranded experiment suggests that the increase in information from the double-strand is valuable at the beginning. Later, however, some interference occurs in the breeding, causing more unfit children to be produced, and therefore slowing convergence of the population. Because the only differences between the single-stranded and double-stranded encodings is the mysterious effect on the two strands by the mutation and crossover operators, these are the prime suspects. Runs were also done with mutation and crossover rates for the single-stranded experiments equal to those of the double-stranded experiment (not shown). The single-stranded experiments performed even better in these cases.

It seems clear that the strong relationship between the double strands causes a problem. A beneficial mutation on one strand, for example, may cause a destructive mutation on the other strand. The genetic algorithm must perform its optimisation with the constraint imposed by the relationship between the strands.

This effect is similar to that described by Kauffman (1993) in his *NK* model of rugged fitness landscapes. Kauffman models genomes consisting of N genes. The fitness contribution of each gene depends on itself and K other genes on the genome. Evolution

of genomes comprises selection and mutation of genes with no recombination. As K , a measure of the epistatic interactions, increases the fitness landscape becomes more rugged. When K is 0, the landscape consists of one global peak. At the other end of the scale, where K is $N-1$, the landscape is fully random, the number of local optima is extremely large and the height of the fitness peaks decreases towards the average fitness. As the number of interactions between genes increases, Kauffman says that “optimisation can attain only ever poorer compromises”. This is exactly the problem we face. Relationships between genes on single-stranded genomes are those imposed by the genomic language and the problem itself. Double-stranded genomes, however, add the additional constraint of the relationship between strands and its effect on the genetic operators.

One way to mitigate the effect of this relationship between genes on separate strands may be to make the strands longer. With longer strands, there is a better possibility of having non-coding information on the strand facing a gene rather than having genes cramped together. This hypothesis warrants further investigation.

The single-strand experiments always seem to perform better than the double-strand experiment. Nature employs a double-stranded coding. Why should it choose an inferior solution? We believe the answer lies in the other use of double-stranded codings, the use that has no analogue in our model. That is, that two strands ensure greater stability of the DNA molecule. Perhaps the interconnectedness of the strands is the price Nature pays for the benefit of molecular stability.

Chapter 11

CONCLUSION

Everything's got a moral, if you can only find it.

- Lewis Carroll, "Alice's Adventures in Wonderland", Ch. 9

This chapter concludes the thesis. It summarises the cell model and the experiments performed in the last two chapters. Following this, it examines some future work as well as some fanciful ways to use the cell model in other areas of computer science.

In this thesis, we presented a novel model of an abstract biological cell that we evolved to adapt to simple environments. The primary features of our cell model are the genome and metabolism. A model of cell membrane is based on the metabolism.

The metabolism is based on chemical kinetic simulations by Farmer, Bagley and others (Farmer, Kauffman, and Packard (1986), Bagley and Farmer (1991), Bagley, Farmer and Fontana (1991)) and results in a system of coupled nonlinear differential equations.

Our genome model is similar to those commonly used in genetic algorithms since we wished to evolve the cells using this class of algorithms. We extended the genome model in two important biologically plausible ways: the addition of two strands; and the introduction of a parallel genomic language that facilitated gene expression and regulation.

There are two benefits of having two strands rather than one: the number of schemata that are represented by each genome is much larger than for single-strand encodings; and a biologically inspired algorithm for the inversion genetic operator is simple to implement. The increase in number of schemata for double-strand encodings gives the

genetic algorithm more information to use and allows faster search of the problem space. Increased rates of inversion were shown to increase the convergence of a population. The drawback of using a double-strand encoding scheme is that the strands impose a further constraint on the genetic algorithm: modification of information on one strand may adversely affect information on the other strand. The constraint decreases the rate of convergence of a population, although this may be mitigated by an increased rate of inversion. Further research is required to determine whether longer genomes may further mitigate this constraint.

Our genome model encodes four bit tokens (or bases) from a custom-built parallel genomic language. The crossover and inversion genetic operators are constrained to cutting the genome only at nibble boundaries. We performed experiments relaxing this constraint to allow crossover and inversion to act between any consecutive bits. Results suggested that this relaxation gave inferior performance because noise was added to the genome whenever crossover or inversion occurred.

The parallel genomic language follows the operon model of Jacob and Monod. It allows gene expression and regulation and encodes genes that code for enzymes or carriers. Genes are regulated in blocks called operons. A genomic language such as ours is more biologically realistic than the simple scheme commonly used in genetic algorithms where the semantics of a gene is determined by its (fixed) loci. Our scheme allows the inversion operator to function because genes are not restricted to fixed loci. The gene expression algorithm is much more biologically plausible than others commonly used in genetic algorithms and it adds additional information that the cell and genetic algorithm can optimise. The semantics of our language impose a relationship between bases that may be sensitive to attacks from mutation. Further research is required to investigate the sensitivity of this relationship.

The genome and chemical metabolism of a cell in our model have a close relationship. The genome specifies families of chemical reactions that are catalysed and families of chemicals that may diffuse through the cell membrane at increased rate. Chemicals produced from metabolic processes regulate genes and allow expression of proteins from the genome. Regulation and expression of genes fits into the system of differential equations in our model, solution of which is achieved using an adaptive Runge-Kutta

numerical integration algorithm. The genome and metabolism must coevolve to produce a cell that can survive in its environment. Evolution of the genome is Darwinian, whereas evolution of the metabolism has Lamarckian features.

We investigated the “bootstrapping” problem. This involved evolving cells that could exist in a simple environment from random genomes and simple initial metabolism. This problem was solvable and a gallery of common cellular strategies was described.

Experiments show that cell populations find solutions to the “bootstrapping” problem much easier with coevolution than when the initial metabolic conditions remain fixed. With the latter, a particular genome must be found to match the metabolic conditions. With coevolution, however, only a match between some initial metabolic conditions and a genome need be found. Because the initial metabolic conditions may change, many more pairs may be found and coevolution can move towards a favourable region of cell space.

Solution to the “bootstrapping” problem was made difficult because the fitness function cannot differentiate between cells exhibiting myopic strategies and those with long-term solutions. Cells with myopic strategies receive high fitness but produce children that have a higher probability of receiving low fitness (ie, when the short-term solution begin to fail). Longer-term strategies usually attain more modest fitness values. A novel solution using population management, where the fitness of parent cells is retroactively modified when the fitness of their offspring becomes known, reduces the number of cells exhibiting myopic strategies.

The fitness function that scores cells is a function of six parameters: changes in cell volume; cell life time; correlation between genome switching regions and switching chemicals; complexity of the metabolism; the number of cell membrane modifying chemicals; and a measure of parsimony in the cell. We found that fitness is more closely correlated with the mother cell than with the father when coevolution is permitted.

Future Research

This project has raised many questions. Consequently, there is a variety of directions for future research.

Research with the Current Model

Due to computational and time constraints, only minimal tuning of parameters could be achieved in this project. The effect of modifying many of the parameters is unknown. These include changing reaction rates, population sizes, transcription rates and protein degradation rates. Interactions between parameters must be determined.

It would be interesting to increase the slope of the sigmoid function determining gene regulation to make gene switching more discrete (step-like). That is, to decrease the amount of blocker or activator required to turn an operon from off to on, or alternatively to increase the time a blocker or activator binds to the promoter region. This is accomplished by modifying the parameter K in equation (6-7). In this project, we aimed for a smoother function to allow evolution a flatter hill to climb. Since the environments appear to be easy for the cell to solve, the effect of more discrete gene switching would be interesting.

Another major set of parameters that can be modified are those controlling facilitated diffusion. As section 10.2 discusses, an increase in the binding, diffusion and unbinding rates for carrier molecules will speed up the action of carriers compared to enzymes. As well, the effect of a different carrier matching rule that would allow more specific matching of small molecules and exclude similar larger molecules may give favourable results.

We have not run many experiments with different lengths of simulation time. Preliminary results (not presented in this thesis) suggest that running simulations for a longer time eliminates more of the cells with myopic strategies. Running cells for longer may also give better results for facilitated diffusion experiments because time is available for the diffusion to occur. As well, longer simulation times will affect the

division of the problem posed by the environment into sub-tasks by a germ line because each cell will be able to solve more of the problem independently. Unfortunately, it is difficult to compare the fitness of cells that were run for different simulation times. Cells that run for a short time get higher fitness values than cells that are simulated for more time. The reason for this is that more cells will survive in a population that runs for a short time than in a population that runs for a longer time. As well, cells that run for only a short time are not able to change their volume as much as cells that run for longer.

Analysis should be performed on experiments that vary the length of genome. With longer genomes, more genes that are useful become possible, but damaging mutations before genes become more likely.

More analysis of double-stranded genomes is warranted. In particular, methods of minimising the interference between the two strands should be investigated. Likely techniques include using longer genomes or different coding schemes. Coding schemes featuring benign symbols that are ignored by the genomic language may be useful. What density of genes can a genome of a particular length sustain before the interference becomes too great? The effect of genetic operators on schemata represented by double-stranded genomes also needs much investigation.

Experiments analysing the sensitivity to mutation of genomes encoding strings from a genomic language would guide us in constructing such languages. In particular, interactions between bases (tokens) implied by the language and the number of bases over which these interactions operate is important.

We have stated that the two environments discussed in chapter 9 appear to be easy for the genetic algorithm to solve. It would be exciting to see what solutions are found for environments that are more complex. Environments that require solutions having more than one operon are particularly enticing. Complex environments would allow further analysis of gene regulation too. Our simple environments didn't much show the capabilities of gene switching. More analysis of the relationship between bit genomes and nibble genomes would be possible: with more difficult problems, does exploitation

of the increased number of schemata for bit genomes outweigh the noise added, compared to nibble genomes?

The major problem faced when analysing the behaviour of cells during their simulation is the vast amount of data produced. There is a concentration value of each species of chemical for every time step. Multiplied by the thousand or so cells produced by the genetic algorithm, this generates too much data to analyse easily. Techniques need to be devised to derive the rules governing each cell (such as those presented as case studies in chapter 9) from the raw data. We need to determine what data can be discarded and what is important. The problem appears to be similar to explaining how artificial neural networks operate.

Part of the problem of understanding how one of the cells operates is analysis of the use of different chemical species to the cell. Can we eliminate chemicals that have no use from the cell or are they a by-product of the metabolism? Are these useless chemicals eventually exploited by evolution?

Another associated question relates to how solution to the problem posed by the environment is broken up by the germ line into chunks that can be solved by individual cells. No foresight is intended with the subdivision of the environmental problem, since evolution is not planned. We are simply interested in how the problem *can* be broken up. Can all environmental problems be broken up? How does the length of simulation time relate to this process? Some ways of subdividing the problem are more useful to evolution than others. Each step must produce cells with high enough fitness to survive and breed. Steps that allow many other different *next* steps (cells solving the next sub-problem) will be particularly profitable. On the other hand, cells that reach a dead-end in solving the problem (that is, those that permit no next steps or myopic cells) aren't beneficial. This idea seems to be analogous to the concept of the "evolution of evolvability" (Dawkins 1988).

Extending the Model

Various ways to extend the model present themselves. The first of these is to change the parallel genomic language to allow an operon to be regulated by both activators and blockers. As the model stands, only one form of regulation is allowed.

Another extension is the introduction of time-varying environments. The aim of these more complex environments is to evolve homeostasis in the cells. The environment would subject the cell to random changes in the amount of chemicals in the world. Cells would be evolved to live under a variety of such stimuli. A stochastic fitness function would be used. A cell would live in a variety of randomly selected environments and its fitness value would be a function of how well it survived in each (possibly weighted by some measure of the violence of the environment's stimuli).

A seemingly unlimited way to extend the model is to make it more biologically plausible. Evolution of a more plausible cell-cycle reproduction algorithm would be interesting. Cells could be given a physical topography complete with cytoskeleton and perhaps cilia and flagella.

Cohabitation of cells in the environment and their interaction and competition for resources would be stimulating. The fitness function would disappear from such a model. An environment with competing cells may need to be seeded with cells known to survive in the isolated environment.

Another way to determine the fitness would be to allow cells to build themselves entirely from substances in their environment. In such a simulation, fitness would be determined implicitly and the modelling of the genome and cellular membrane would be somewhat different. A simulation such as this would be similar in spirit to Echo model (Holland 1995). This was not attempted in the current model because we deemed it to be computationally infeasible.

The mechanism of retroactive change in parent's fitness could also be removed from the simulation. Nature uses r-selection to select simple organisms living in complex

environments (Solomon et al. 1993). Many of the characteristics of r-selected organisms apply to our simulated cells. For example, they have small physical size, short life expectancy and early maturity. They also produce many young, most of which die before maturity. The latter characteristic of large families leads us to guess that larger populations and larger families might solve the problem of myopic strategies.

One of the most exciting areas of research in such a model would be analysis of the primitive communication protocols that emerge between cells in the population. Other advanced work could model assemblies of cells with a view to evolution of cooperative behaviour. The biological analogies for these assemblies are tissue, an organ or multicellular organisms. Such experiments would introduce models of differentiation.

Related Future Work

It is interesting to reflect on the applicability of our cell model to other problems in computer science. Parallel genomic languages appear to be useful for representing semantics on a genome. They have been used in Holland's broadcast language (Holland 1992) and in classifier systems (Holland 1992, Goldberg 1989). They allow genes to become disassociated from their locus and allow the inversion genetic operator to be implemented easily. The basic problem faced by the cell is one of self-control and homeostasis. Many other systems try to solve these problems as well. An approach using a genomic language similar to ours, but with a different model of metabolism may be appropriate. This, again, is a similar idea to classifier systems.

A final tantalising idea is to model parallel computer programs using a symbolic analogue to the cell model. The motivation would be to replicate the robustness of cells: computer programs are notoriously sensitive to small changes in code. Each cell would become an object in the computer program (an object-oriented program). The genome would still stand, but expressed proteins could become subprograms (subroutines) or perhaps the triggers for subroutines (metabolic pathways). Chemical species become member variables of the object (or attributes of the object). The differential equations and their variables would disappear, to be replaced with a blackboard memory of variables local to the object (member variables). The cell membrane becomes an

interface to the object, allowing only a small variety of calls (diffusion of a small family of chemicals). The idea of ontogeny becomes execution of the program: objects divide into other objects and send messages (chemical signals) to one another.

Glossary

ADP: adenosine diphosphate. A molecule used for energy transfer in cells. This is the 'discharged' state.

allele: One of a number of variations of a gene coding for a particular characteristic that occupy the same locus on the chromosome.

allosteric modification of enzymes: The binding of a molecule of a substance (other than the normal substrate) to a site on an enzyme that causes the enzyme to change shape and activity.

amino acid: An organic molecule used to build the peptide chains of proteins.

ATP: adenosine triphosphate. A molecule used for energy transfer in cells. This is the 'energised' state.

autocatalytic reaction network: A set of polymers where each polymer is produced by reactions involving other polymers in the set. These reactions are catalysed by polymers in the set.

base: One of five kinds of molecule (adenine, guanine, cytosine, thymine or uracil) that is part of a nucleotide molecule that makes up DNA or RNA.

bootstrapping problem: The problem of coevolving a genome and metabolism from random beginnings to produce a model of a cell that acts as a coherent union and that can survive in a simulated environment.

broadcast language: The language designed by Holland to be encoded on an artificial genome that consists of a set of production rules.

canalyzing Boolean function: “Any Boolean function having the property that it has at least one value (1 or 0) which suffices to guarantee that the regulated element assumes a specific value (1 or 0)” (Kauffman 1993)

carrier molecule: A molecule in the simulation that transports a molecule across the semipermeable membrane.

chemical concentration: A real value in the interval $[0, 1]$ denoting the relative amount of that chemical in the cell.

chemical species: A group of chemicals with the same shape.

chloroplast: An organelle of some plant cells that is the site of photosynthesis.

cleavage reaction: A chemical reaction that cuts a polymer into two smaller molecules.

codominant alleles: “Alleles that produce independent effects when heterozygous” (Gardner, Simmons and Snustad 1991)

codon: A sequence of three bases that specify an amino acid.

coevolution: “The interdependent evolution of two or more species that occurs as a result of their interactions over a long period of time”. (Solomon et al. 1993)

complementary matching: The matching of bases where adenine matches thymine and guanine matches cytosine. A similar matching occurs in artificial genomes where 0 matches 1.

condensation reaction: A chemical reaction that combines two polymers into one longer polymer.

constitutive operon: An operon that expresses all the time.

crossover: The recombination of two chromosomes resulting in the exchange of genetic material.

differentiation: The process whereby a young, unspecialised cell changes into a more specialised cell.

diploid: The state where two sets of chromosomes appear in a cell.

DNA molecule: deoxyribonucleic acid. This molecule stores the genetic information of a cell.

DNA transcription: The synthesis of RNA from DNA.

dominant allele: “The allele that is always expressed when present, regardless of whether homozygous or heterozygous.” (Solomon et al. 1993)

elitist strategy: A strategy used in genetic algorithms whereby fit members of the population are maintained in later generations.

enzyme: A protein that catalyses a chemical reaction.

epistatic interaction: A dependency such that “the fitness contribution of one or another allele of one gene may often depend upon the alleles of some of the remaining ... genes”. (Kauffman 1993) That is, nonlinear interactions between genes.

eukaryotic cell: A cell containing a nucleus and other membrane-bounded organelles.

facilitated diffusion: The transport of a molecule across the semipermeable membrane of the cell using another chemical.

family of chemicals: A set of species of chemicals that are similar (but not exactly the same) in shape.

gene: A sequence of bases that specify a protein.

gene expression: The process of protein production in our model. It combines transcription and translation.

gene regulation: In our model, the process of changing the activation of genes using other activator or repressor molecules.

genetic operator: One of the three operations (inversion, crossover or mutation) that are applied to genomes in a genetic algorithm.

genome: In our model, the genetic material of the cell.

genotype: “The genetic makeup of an individual”. (Solomon et al. 1993)

germ line: In our model, we mean this to be the ancestors of a given cell.

Gray code: An encoding scheme for integers in genetic algorithms such that a one bit mutation to the genome changes the coded integer by only 1.

haploid: The state where one set of chromosomes appear in a cell.

heterozygous: Where two alleles at a locus are unlike.

hierarchy of automata: A sequence of finite state machines where each state may contain a finite state automata.

homeostasis: The maintenance of a constant internal environment in a system such as a cell or organism.

homozygous: Where two alleles at a locus are like.

inducible operon: An operon that defaults to not expressing genes. An activator molecule must be present to allow gene expression.

inversion: A mutation of a chromosome where a part of the chromosome is turned end for end and reattached to the same chromosome.

locus: The position on a chromosome where a gene for a given trait occurs.

lumping species: A modelling technique used by Weinberg (1970) where a number of chemical species are aggregated.

metabolic network: An autocatalytic reaction network where much of the mass of its environment is reduced into a relative few species. (See Bagley and Farmer 1991).

metabolism: In this model, the chemical reactions that produce and maintain a cell.

mitochondria: An intracellular organelle that is the site of oxidative phosphorylation. It produces energy for the cell in the form of ATP.

mRNA: A messenger RNA molecule transcribes genes from the chromosome and is read by ribosomes.

mutation: A change in a chromosome. In genetic algorithms, a mutation is a bit change of the genome.

myopic strategy: In our model, a strategy employed by a cell that does not succeed in the longer term.

nucleotide: One of the molecules making up nucleic acid polymers such as RNA or DNA.

ontogeny: The developmental history of an organism.

operon: The smallest sequence of a genome that can be regulated in prokaryote cells. It comprises a promoter region and a sequence of genes.

organelle: A specialised structure within a cell such a ribosome, mitochondria, chloroplast.

osmosis: Diffusion of water through a membrane so that the concentration of ions in both regions is balanced.

parallel genomic language: In this model, the language encoded on the genome of a cell.

permeant molecule: In this model, a molecule that diffuses through the semipermeable membrane.

phenotype: “The physical or chemical expression of an organism’s genes”. (Solomon et al. 1993)

polymer: A molecule composed of repeating molecules of a similar type.

polypeptide: A chain of amino acids linked by peptide bonds.

prokaryote cells: A cell that does not contain a nucleus or other membrane-bounded organelles. eg, bacteria

promoter region: The site on DNA where expression of operons may be initiated.

protein: A polymer consisting of many amino acids. Proteins are the main structural units in cells.

proto-genetic algorithm: By this we mean a genetic algorithm that was devised before the ‘accepted’ style had crystallised.

protozoa: A single-celled eukaryote organism.

recessive allele: An allele that is not expressed in the heterozygous state.

repressible operon: An operon that defaults to expressing genes. A blocker or repressor molecule must be present to stop gene expression.

ribosome: An organelle that reads a codon from a mRNA molecule and binds the specified nucleic acid to a growing protein molecule.

roulette wheel selection: A method used in genetic algorithms of selecting parents from a population where the probability of choosing a member is proportional to its fitness.

semidominant alleles: Alleles that “produce the same product but in lesser quantity as compared with the dominant allele”. (Gardner, Simmons and Snustad 1991)

semipermeable membrane: A membrane that allows some permeants to pass through more easily than others.

spontaneous reaction: In this model, a chemical reaction that occurs without catalysis.

substrate: A substance on which an enzymes operates.

translation: The synthesis of proteins as specified by RNA.

tRNA: A transfer RNA molecule is used by the ribosome to match a codon to an amino acid.

Appendix A

Derivation of $\hat{\rho}_n$ the probability that one of n competing switch chemicals is bound to the promoter region of an operon.

We will initially derive $\hat{\rho}_n$ from first principles in a recursive style. Firstly, observe that

$$\hat{\rho}_{n+1} = \frac{\sum_{i=1}^n T_i + T_{n+1}}{T} \quad (\text{A-1})$$

where T_i is the time that the i th switch chemical is bound to the promoter region and T is the total amount of time of interest. This can be rewritten as

$$\hat{\rho}_{n+1} T = \sum_{i=1}^n T_i + T_{n+1} \quad (\text{A-2})$$

That is, the total amount of time multiplied by the probability of any of $n+1$ chemicals being bound is equal to the amount of time that each of the switch chemicals is bound. The probability is a throughput.

As well,

$$\sum_{i=1}^n T_i = \hat{\rho}_n (T - T_{n+1}) \quad (\text{A-3})$$

$$T = \frac{\sum_{i=1}^n T_i}{\hat{\rho}_n} + T_{n+1} \quad (\text{A-4})$$

Similarly, looking at only the probability of the $n+1$ st switch chemical binding to the promoter, we can state

$$T_{n+1} = \rho_{n+1} \left(T - \sum_{i=1}^n T_i \right) \quad (\text{A-5})$$

$$T = \sum_{i=1}^n T_i + \frac{T_{n+1}}{\rho_{n+1}} \quad (\text{A-6})$$

Solving the simultaneous equations (A-4) and (A-6) by subtracting (A-6) from (A-4), we get

$$0 = \frac{\sum_{i=1}^n T_i}{\hat{\rho}_n} - \sum_{i=1}^n T_i + T_{n+1} - \frac{T_{n+1}}{\rho_{n+1}}, \quad (\text{A-7})$$

which simplifies to give

$$T_{n+1} = \frac{\left(\sum_{i=1}^n T_i\right)\rho_{n+1}(1-\hat{\rho}_n)}{\hat{\rho}_n(1-\rho_{n+1})} \quad (\text{A-8})$$

Substituting (A-8) into (A-4) yields

$$T = \frac{\sum_{i=1}^n T_i}{\hat{\rho}_n} + \frac{\left(\sum_{i=1}^n T_i\right)\rho_{n+1}(1-\hat{\rho}_n)}{\hat{\rho}_n(1-\rho_{n+1})} \quad (\text{A-9})$$

Substitute (A-8) into (A-2)

$$\hat{\rho}_{n+1}T = \left(\sum_{i=1}^n T_i\right)\left(1 + \frac{\rho_{n+1}(1-\hat{\rho}_n)}{\hat{\rho}_n(1-\rho_{n+1})}\right) \quad (\text{A-10})$$

Finally, substitute (A-9) into (A-10) and simplify to get

$$\hat{\rho}_{n+1} = \frac{\frac{\hat{\rho}_n}{1-\hat{\rho}_n} + \frac{\rho_{n+1}}{1-\rho_{n+1}}}{\frac{1}{1-\hat{\rho}_n} + \frac{\rho_{n+1}}{1-\rho_{n+1}}} \quad (\text{A-11})$$

We believe, from generalisation, that an iterative version of $\hat{\rho}_n$ exists and that its definition is

$$\hat{\rho}_n = \frac{\rho_1 + (1-\rho_1)I(n)}{1 + (1-\rho_1)I(n)}, \quad (\text{A-12})$$

$$I(n) = \sum_{i=2}^n \frac{\rho_i}{1-\rho_i}$$

(A-12) can be proved by induction using the recursive form (A-11). We will proceed to do this now.

I. Show that (A-12) is valid for $n = 1$

$$\hat{\rho}_1 = \frac{\rho_1 + (1 - \rho_1)I(1)}{1 + (1 - \rho_1)I(1)} \quad (\text{A-13})$$

$$= \frac{\rho_1 + (1 - \rho_1)0}{1 + (1 - \rho_1)0}$$

$$= \rho_1$$

II. Assume that $\hat{\rho}_i$ is valid.

$$\hat{\rho}_i = \frac{\rho_1 + (1 - \rho_1)I(i)}{1 + (1 - \rho_1)I(i)} \quad (\text{A-14})$$

III. Show that $\hat{\rho}_{i+1}$ follows from $\hat{\rho}_i$ above, using the recursive form (A-11).

$$\hat{\rho}_{i+1} = \frac{\frac{\hat{\rho}_i}{1 - \hat{\rho}_i} + \frac{\rho_{i+1}}{1 - \rho_{i+1}}}{\frac{1}{1 - \hat{\rho}_i} + \frac{\rho_{i+1}}{1 - \rho_{i+1}}} \quad (\text{A-15})$$

$$\begin{aligned} & \frac{\left(\frac{\rho_1 + (1 - \rho_1)I(i)}{1 + (1 - \rho_1)I(i)} \right)}{\left(1 - \frac{\rho_1 + (1 - \rho_1)I(i)}{1 + (1 - \rho_1)I(i)} \right)} + \frac{\rho_{i+1}}{1 - \rho_{i+1}} \\ &= \frac{1}{1 - \frac{\rho_1 + (1 - \rho_1)I(i)}{1 + (1 - \rho_1)I(i)}} + \frac{\rho_{i+1}}{1 - \rho_{i+1}} \end{aligned} \quad (\text{A-16})$$

This simplifies to give

$$\begin{aligned} \hat{\rho}_{i+1} &= \frac{\rho_1 + (1 - \rho_1) \left(I(i) + \frac{\rho_{i+1}}{1 - \rho_{i+1}} \right)}{1 + (1 - \rho_1) \left(I(i) + \frac{\rho_{i+1}}{1 - \rho_{i+1}} \right)} \\ &= \frac{\rho_1 + (1 - \rho_1)I(i+1)}{1 + (1 - \rho_1)I(i+1)} \end{aligned} \quad (\text{A-17})$$

So, III follows and

$$\hat{\rho}_n = \frac{\rho_1 + (1 - \rho_1)I(n)}{1 + (1 - \rho_1)I(n)} \quad (\text{A-18})$$

is valid for all n .

Derivation of $d\hat{\rho}_n/dt$.

We need the derivative of $\hat{\rho}_n$ for the system of differential equations. Each ρ_i in $\hat{\rho}_n$ is a function of the concentration of the switch chemical it describes.

$$\rho_i = 1 - e^{-K_i s_i} \quad (\text{A-19})$$

We can find the derivative with respect to time using the chain rule.

$$\begin{aligned} \frac{d\rho_i}{dt} &= \frac{d\rho_i}{ds_i} \frac{ds_i}{dt} \\ &= K_i e^{-K_i s_i} \frac{ds_i}{dt} \end{aligned} \quad (\text{A-20})$$

We already know the value of ds_i/dt . They are our differential equations. Using the chain rule again, we can find

$$\frac{d\hat{\rho}_n}{dt} = \sum_{i=1}^n \frac{\partial \hat{\rho}_n}{\partial \rho_i} \frac{d\rho_i}{dt} \quad (\text{A-21})$$

This simplifies to give

$$\frac{d\hat{\rho}_n}{dt} = \left(\frac{1 - \rho_1}{1 + (1 - \rho_1)I(n)} \right)^2 \sum_{i=1}^n \left(\frac{1}{(1 - \rho_i)^2} \frac{d\rho_i}{dt} \right) \quad (\text{A-22})$$

Appendix B

Derivation of membrane formulae.

At the outset, we make a number of assumptions about the cell.

1. The cell is spherical and filled with as much water as the membrane will hold.
2. Trans membrane carrier molecules do not significantly increase the surface area of the cell.
3. Cell membrane molecules are particulate spheres. This is most definitely different to natural cells.

B.1 Calculation of the Width of the Cell Membrane

We model the membrane as a sphere packing (Kittel 1971) of cell membrane molecules. Two layers of spheres are necessary as illustrated in figure B.1.

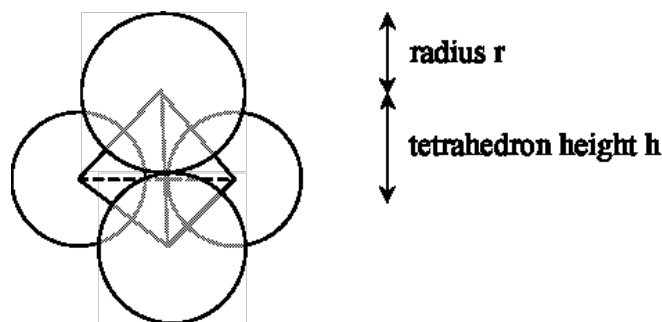


Figure B.1 cell membrane sphere packing

The solid with vertices at each molecule's centre is a tetrahedron. The width of the membrane, then, is

$$\text{width of membrane} = 2r + h \quad (\text{B-1})$$

where r is the radius of the sphere making up the membrane molecule, and h is the height of the tetrahedon. Using trigonometry we can determine the width of the membrane as

$$w = 2r + 2r\sqrt{2/3} = 2r\left(1 + \sqrt{2/3}\right) \quad (\text{B-2})$$

Typically, we use $r = 4.4 \times 10^{-9}$ m.

B.2 Calculation of the Volume of the Cell

The number of membrane molecules covering the cell can be expressed as

$$N_M = 0.74 \frac{V_W}{V_1} \cong \frac{3V_W}{4V_1} \quad (\text{B-3})$$

where

N_M is the number of membrane molecules in the cell,
 0.74 is the packing density of the spheres (Kittel 1971),
 V_W is the volume of the cell wall, and
 V_1 is the volume of one membrane molecule.

We use 3/4 as an approximation to 0.74. This implies the packing of slightly squishy spheres.

Reorganising, we get

$$V_W = \frac{4V_1 N_M}{3} \quad (\text{B-4})$$

Another way to define the volume of the wall is to multiply the surface area of the cell with the width of the membrane. This assumes that the cell wall is a flat surface, ignoring curving of the membrane.

$$V_W = 4\pi R^2 w \quad (\text{B-5})$$

where R is the radius of the cell. Substituting (B-2) into (B-5) we get

$$V_W = 8\pi R^2 r \left(1 + \sqrt{2/3}\right) \quad (\text{B-6})$$

Equating (B-4) and (B-5), we get

$$8\pi R^2 r \left(1 + \sqrt{2/3}\right) = \frac{4V_1 N_M}{3} \quad (\text{B-7})$$

Since $V_1 = \frac{4}{3}\pi r^3$, this simplifies to

$$\pi R^2 r \left(1 + \sqrt{2/3}\right) = \frac{2\pi r^3 N_M}{9} \quad (\text{B-8})$$

$$R^2 = \frac{2\pi r^2 N_M}{9(1 + \sqrt{2/3})} \quad (\text{B-9})$$

The volume inside the cell is given by

$$V_{IN} = \frac{4}{3}\pi R^3 \quad (\text{B-10})$$

and, using (B-9), this yields

$$V_{IN} = \frac{4}{3}\pi \left(\frac{2\pi r^2 N_M}{9(1 + \sqrt{2/3})} \right)^{3/2} \quad (\text{B-11})$$

B.3 Calculation of the Surface Area of the Cell

$$A = 4\pi R^2 \quad (\text{B-12})$$

$$A = 4\pi \frac{2\pi r^2 N_M}{9(1 + \sqrt{2/3})} \quad (\text{B-13})$$

$$A = \frac{8\pi^2 r^2 N_M}{9(1 + \sqrt{2/3})} \quad (\text{B-14})$$

B.4 Calculation of the Number of Water Molecules in the Cell

1 m³ of water has mass 10⁶ g.

This means the mass of water in the cell is 10⁶ V_{IN} g.

The atomic mass of water is readily calculated from the atomic masses of hydrogen and oxygen.

H has atomic mass 1.0079 daltons.

O has atomic mass 15.9994 daltons.

The atomic mass of H₂O is 18.0152 daltons. Call this constant A_w.

The number of molecules in A_w grams of water is N_A, Avogadro's number (6.0220e+23 mol⁻¹).

Using (B-11) above, then, the number of water molecules in the cell is

$$\begin{aligned} N_{WC} &= \frac{N_A}{A_w} 10^6 V_{IN} \quad (\text{B-15}) \\ &= \frac{N_A}{A_w} 10^6 \frac{4}{3} \pi \left(\frac{2\pi r^2 N_M}{9(1 + \sqrt{2/3})} \right)^{3/2} \end{aligned}$$

Appendix C

Estimation of change in cell volume with no metabolic activity

Derivation of the estimate of the change in volume of the cell assuming no metabolic action in the cell.

We wish to calculate an estimate of the change in volume that would occur in the cell if no metabolic activity apart from background diffusion were to occur. In order to determine this we need to calculate the volume of the cell given no metabolic action. From B-11 above, we know that this will be

$$V_{NA} = \frac{4}{3} \pi \left(\frac{2\pi r^2 N_{MNA}}{9(1 + \sqrt{2/3})} \right)^{3/2} \quad (C-1)$$

where V_{NA} is the volume with no activity and N_{MNA} is the number of membrane molecules given no metabolic activity.

The number of membrane molecules in the cell with no metabolic action is calculated by integrating the standard differential equation we have defined for the change in the number of wall molecules with respect to time.

$$\frac{dN_{MNA}}{dt} = N_{WC} \left(k_1 \sum (\text{builder concs}) - k_2 \sum (\text{breaker concs}) \right) \quad (C-2)$$

where N_{WC} is a constant value. We typically choose the number of water molecules at time 0. Using a constant value here is the reason we're calculating an estimate. As well, we need to integrate each of the differential equations for the builder and breaker molecule concentrations. Because we are no longer interested in reactions and the like, these differential equations are much simpler. They are all of the form:

$$\frac{dc}{dt} = -K_D (c - c_{out}) \quad (C-3)$$

where c is the concentration of the chemical species, c_{out} is the concentration of the chemical species in the environment, and K_D is the diffusion constant defined in section 7.3 (equations 7-7 and 7-8)

$$K_D = K \frac{8\pi^2 r^2 N_M}{9(1 + \sqrt{2/3})l^3} \quad (C-4)$$

Again, N_M is a constant. The differential equation (C-3) is readily soluble. With the initial boundary condition, that $c(0) = c_0$, the solution is

$$c = c_{out} + (c_0 - c_{out})e^{-K_D t} \quad (C-5)$$

Integrating (C-2), we find

$$N_{MNA} = N_{WC} \left(k_1 \sum_i (\int a_i dt) - k_2 \sum_j (\int b_j dt) \right) + C \quad (\text{C-6})$$

where C is the constant of integration. Using the initial boundary condition that, at time 0, the number of wall molecules in the cell is N_{M0} and that $\int c dt = c_0$ we find

$$C = N_{M0} - N_{WC} \left(k_1 \sum_i a_{i0} - k_2 \sum_j b_{j0} \right) \quad (\text{C-7})$$

and substituting in,

$$N_{MNA} = N_{M0} + N_{WC} \left(k_1 \sum_i (\int a_i dt - a_{i0}) - k_2 \sum_j (\int b_j dt - b_{j0}) \right) \quad (\text{C-8})$$

Since the integral of (C-5) is

$$\int c dt = c_{out} t + \frac{c_0 - c_{out}}{K_D} (1 - e^{-K_D t}) + c_0, \quad (\text{C-9})$$

with the initial boundary condition of $\int c dt = c_0$ at time 0, (C-8) simplifies to

$$N_{MNA} = N_{M0} + N_{WC} \left(k_1 \sum_i A_i - k_2 \sum_j B_j \right) \quad (\text{C-10})$$

with

$$A_i = a_{iout} t + \frac{a_{i0} - a_{iout}}{K_{Di}} (1 - e^{-K_{Di} t})$$

and B_j is defined similarly.

Appendix D

Object oriented design of program

Class Diagrams

Class diagrams are given in Booch notation (Booch 1994). Figure D.1 presents a legend for reading the diagrams following. Figures D.2 and D.3 show diagrams for each of the two main programs making up the simulation. The former diagram illustrates the main genetic algorithm driver program (GACell). Figure D.3 illustrates the simulation program for an experiment. That is, for a particular cell. Figure D.4 presents the inheritance hierarchy for haploids and chromosomes and shows how they interact. Figure D.5 shows the Chemical class, which acts as a variable in our simulations, and maps the inheritance hierarchy of ChemicalRep. The relationship between Chemical and ChemicalRep is that of an envelope / letter class. Finally, in figure D.6, the differential equation class (CellDiffEqu) and its family of 'rep' classes (DERep and children) are illustrated.

Relationship of Classes to the Body of the Thesis

The GACell program which runs the genetic algorithm part of the simulation is composed of the classes: GACell, Population, Experiment, ProcessorPool, and Processor. The relationship between these classes is shown in figure D.2. The program is described in chapter 8. As well, the genetic algorithm is described in section 2.4.

The other program written as part of this work, called CellSim, is composed of many more classes. The main high level classes and their relationships are shown in figure D.3. The main algorithm for CellSim is discussed in section 8.1. This section also describes much of the inner workings of the Cell class. The World class which represents the environment of the cell is described in chapter 7. The Vessel class is a simple abstract base class that models common behaviour between the Cell and World classes. CellChromo represents the cell's genome. The functionality of this class (ie, the genome structure and the parsing algorithm) is described in chapter 5. A list of ParsedGene results from parsing the CellChromo. Functionality in ParsedGene is described in chapter 6 (gene expression and regulation). Parts of the Chemical classes which represent the variables of the differential equations and the CellDiffEqu classes are described in many chapters of the thesis.

The Chemical classes, which represent the variables used in the differential equations and the CellDiffEqu classes (that model the differential equations themselves) are described over many chapters. In general, there is a separate class for each of the kinds of variable and a matching class for the differential equation. As well, there is an abstract base class (AbstractChemicalRep) that specifies the interface of each of the chemical classes and a matching class (DERep) for the interface for each of the differential equations. The high level Chemical and AbstractChemicalRep classes are specified in section 3.3. Ordinary chemicals are specified in the classes OrdChemRep

and OrdChemDERep and described in chapter 6 (spiders and gene expression and regulation), chapter 4 (metabolic reactions and protein degradation) and chapter 7 (diffusion and membrane building/breaking). The gene expression and regulation functionality (chapter 6) applies to ordinary chemicals and enzymes alike, so differential equation functionality appears in the BagleyDERep class. Enzymes are specified in the classes EnzymeRep and EnzymeDERep, which are described in chapters 6 (gene expression and regulation) and 4 (reaction catalysis and protein degradation). Carrier molecules are specified by the classes CarrierRep and CarrierDERep and chapters 6 (gene expression) and 4 (protein degradation). Bound forms of these molecules are modelled by the BoundCarrierRep and BoundCarrierDERep classes. These are described in chapter 7 (facilitated diffusion). Expression of proteins using “partly expressed proteins” is described in chapter 6 (gene expression) and modelled by the classes PartlyExpRep and PartlyExpDERep. Gene regulation is modelled by GeneSwitchRep and GeneSwitchDERep and is described in chapter 6. WallRep and WallDERep model the cell membrane and are described in chapter 7. Water molecules in the cell are modelled by WallRep and WallDERep. This is described in chapter 7 (cell membrane) and chapter 4 (metabolic reactions). Terms in the differential equations are modelled using the same class for all equations: CellReactTerm.

The classes modelling the genome are Haploid, BitHaploid, DoubleStrandBitHaploid, DoubleStrandNibbleHaploid, NibbleMixin, Chromosome, CellChromo and ParsedGene. The functionality in these classes is described in chapter 5.

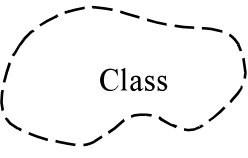

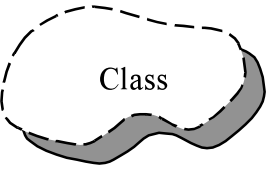
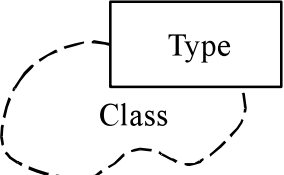


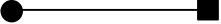
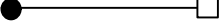
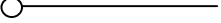
 <p>Class</p>	<p>Class. Abstract class if it contains a </p>
 <p>Class</p>	<p>Class utility. That is, free functions.</p>
 <p>Type</p> <p>Class</p>	<p>Class parameterised on Type</p>
    	<p>Association</p> <p>Inheritance (points to superclass)</p> <p>Aggregation by value</p> <p>Aggregation by reference</p> <p>Uses (client-server)</p>

Figure D.1: Legend of chart notation.

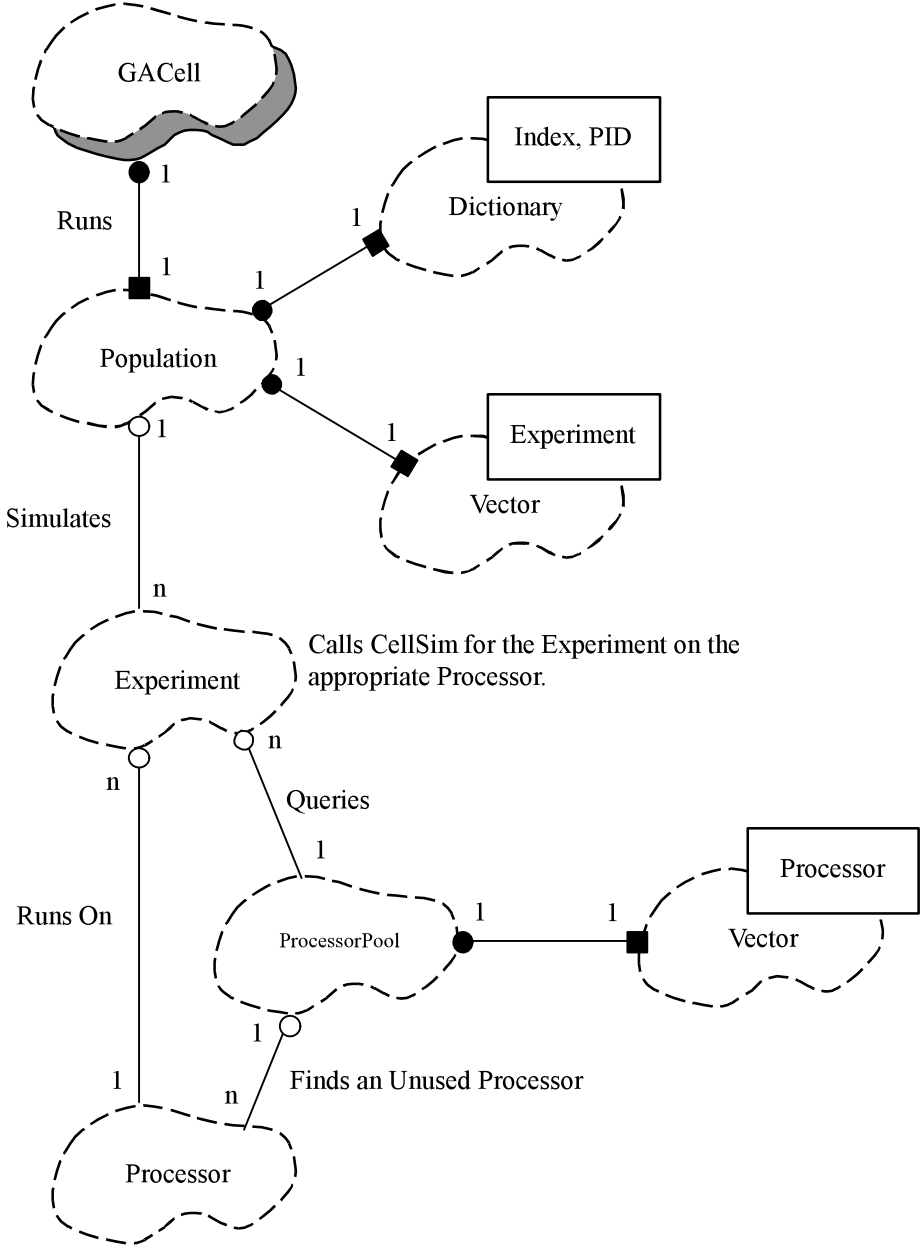


Figure D.2: Chart for Genetic Algorithm Driver Program

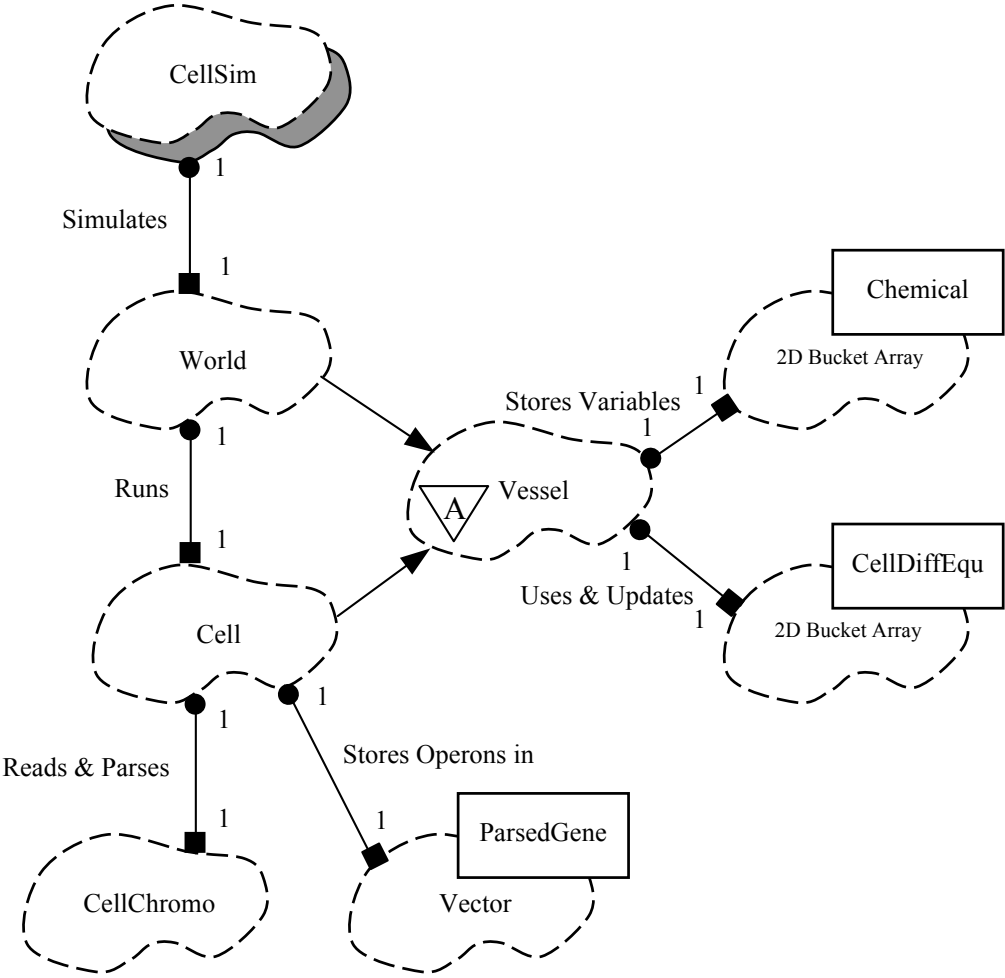


Figure D.3: Chart for Cell Simulation Program

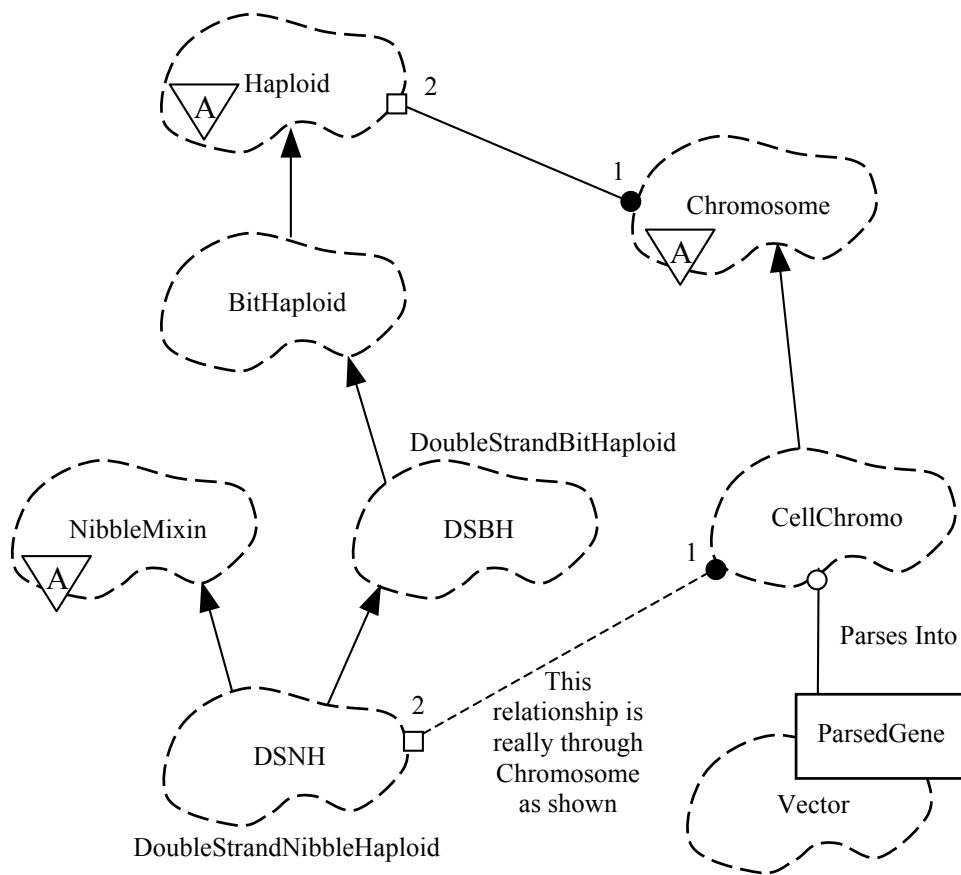


Figure D.4: Chromosomes and Haploids

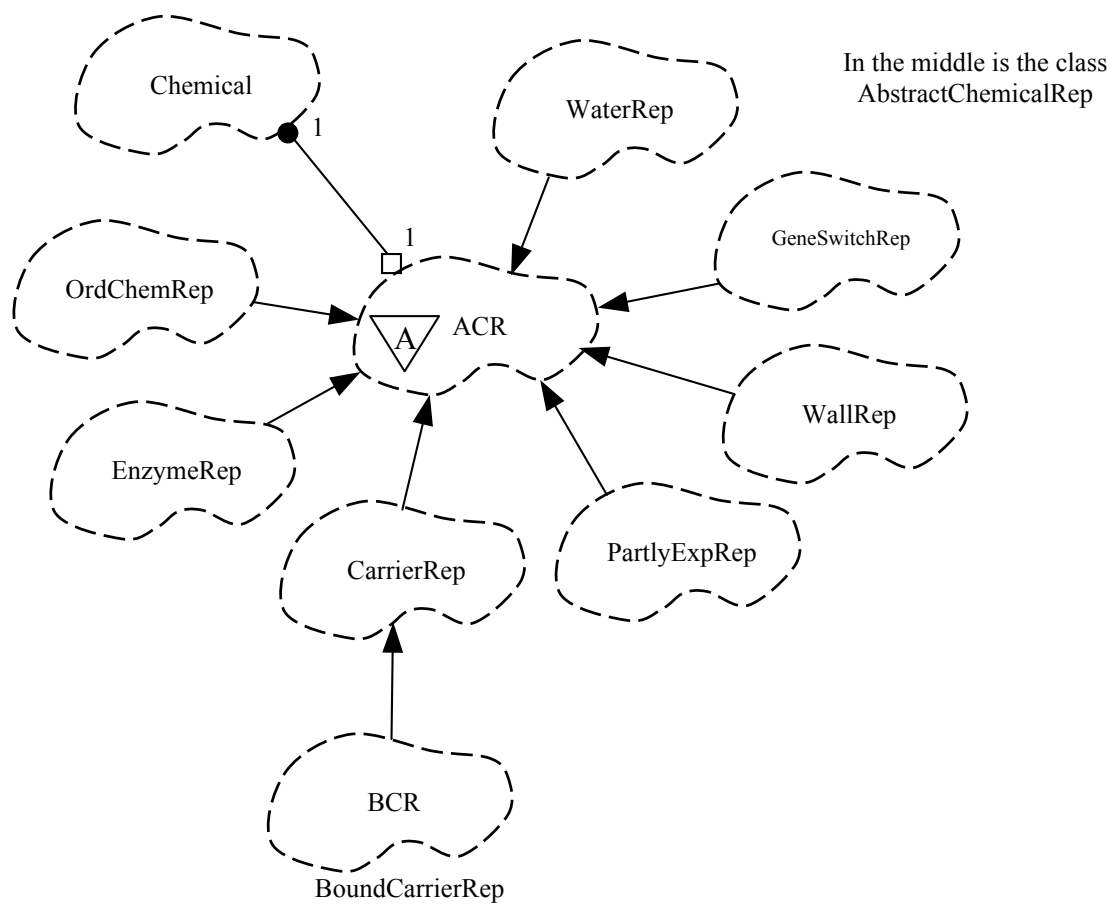


Figure D.5: Chemicals and ChemicalReps

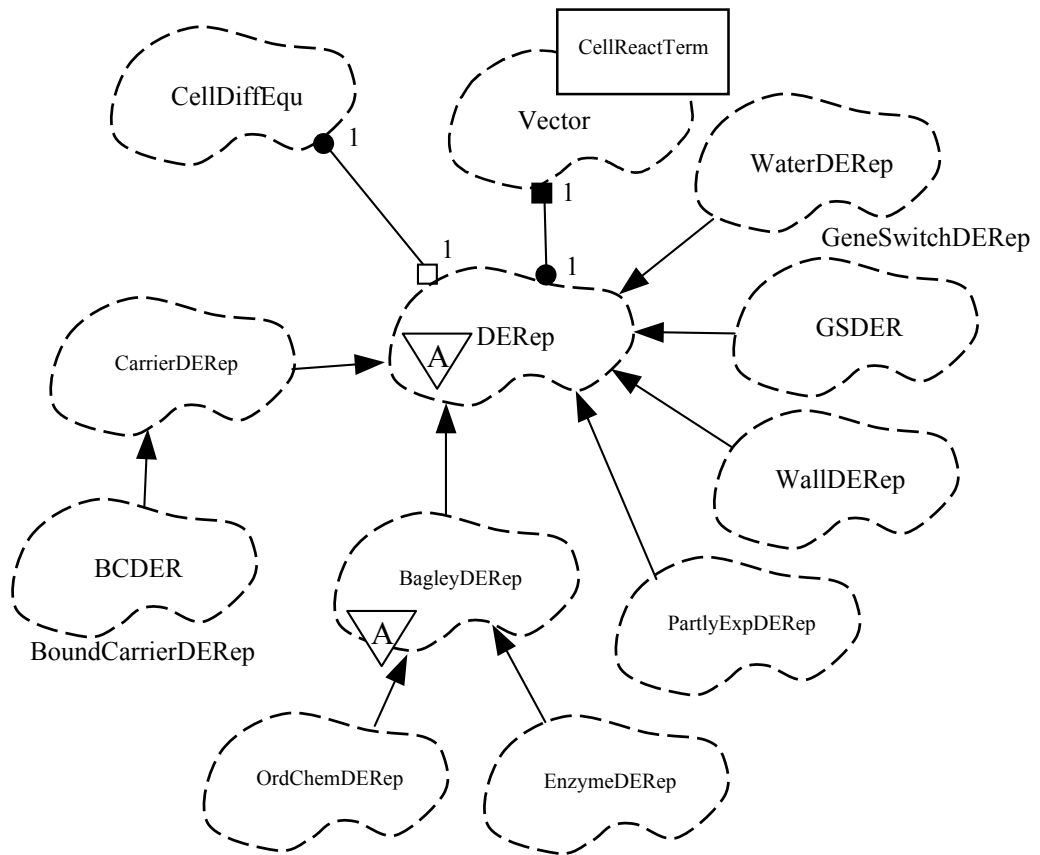


Figure D.6: CellDiffEqs and DEReps

Program Specifications

Program specifications are given in a format suggested by Booch (1994).

Unless otherwise specified, all classes have a copy constructor, assignment operator, destructor and a Dump method (`string Dump() const`). The latter method returns a textual description of the class suitable for printing to the screen or writing to a log file. As well, equivalence operators (`==`) and less than operators (`<`) were written for classes that required them (eg, for sorting objects in a sorted list).

A number of subsidiary classes such as a vector, bit vector, ordered list, dictionary (associative array) and a random number generating class are not given.

There are specifications for two programs. GACell is the genetic algorithm program, which spawns cell simulations (cellsim) on a network of machines. For each program, each component class is given with a short description and a list of the operations able to be performed on / by the class.

GACell

Name: GACell

Definition: Free program

Runs the genetic algorithm evolution of cell simulations

1. Seeds the random number generator
2. Constructs and Run(s) a Population

Population

The Population class holds a number of Experiments and runs them on a variety of Processors.

```
class Population
{
    OVector<Experiment> m_exps; // the experiments
    ODictionary<pid_t, oindex> m_PIDmap; // PID:expIdx map
    OVector<oindex> m_waitQueue; // queue of waiting experiments
    int m_birthCount; // num breedings
    int m_numToBreed; // max num breedings
};
```

Population()

Definition: Default constructor

Looks for a restart file containing experiment ids and fitnesses. If there is one, fill the population vector with these experiments. Any remaining members of the population should be bred from the current population. If there wasn't, look for a seed file containing ids of experiments to seed the population. If this seed file exists, fill the population with these experiments. In any case, fill the rest of the population with random experiments (ie, experiments with cells with random genomes)

void Run()

Definition: Runs the experiments in the population and breeds and runs new experiments until a given number of breeding have occurred.

void RunWaitingExperiments()

Definition: Run as many waiting experiments as possible. That is, until there are no more processors available. Add a map entry for those that were run.

```
void ReportFitness() const
```

Definition: Write information to the log file about the status of the run. Write the number bred, the number to breed, average fitness, highest fitness in the population and the id of the experiment with the highest fitness in the population.

Experiment

Represents a single cell simulation experiment to be run remotely on another machine. Constructs experiments with a random genome for the cell, breeds from two parent cells, or reads from a file (exp.id where id is the experiment's integer identifier). Writes an experiment to its exp.id file. Runs an experiment on a remote machine. Reads the results of an experiment run. Determines whether an experiment has aborted.

```
class Experiment
{
    enum Stat { waiting, running, finished };

    unsigned long m_id;
    Stat m_status;
    CellChromo m_genome;
    double m_fitness;
    double m_avgStepSize; // average step size for numerical integr'n
    double m_lastTime; // time simulation ran till
    unsigned long m_mum;
    unsigned long m_dad;
    string m_procName;
    pid_t m_pid;
    bool m_aborted;
    bool m_resubmitted;
    unsigned long m_age; // time spent in breeding population
};
```

```
Experiment()
```

Definition: Default constructor creates an experiment with a cell containing a blank genome.

```
Experiment(ORNG const &)
```

Definition: Creates an Experiment with a cell containing a random genome. The ID of the experiment is the next unused ID number. This number is managed using the file gastat. This file contains the last used ID. gastat is read by the constructor and the number in it updated. The status of the experiment is set to 'waiting', the genome to a random set of bits, and the other members to their initial values. The exp.id file is created.

```
Experiment(Experiment const & mum, Experiment const & dad)
```

Definition: Creates an Experiment bred from two other experiments. The ID of the experiment is the next unused ID number. The genome is bred from the genomes in each of the parent experiments. The exp.id file is created.

```
Experiment(unsigned long id)
```

Definition: Creates an Experiment by reading its information from an existing exp.id file. The ID is set to the ID of the file and members are initialised according to the file.

```
unsigned long ID() const
```

Definition: Selector. Returns the ID of the experiment.

```
Stat Status() const
```

Definition: Selector. Returns the status of the experiment: waiting, running or finished.

```
double Fitness() const
```

Definition: Selector. Returns the fitness of the experiment, or 0 if the experiment hasn't yet run.


```
double AvgStepSize() const
```

Definition: Selector. Returns the average step size used during the numerical integration of the DEs in the experiment, or 0 if the experiment hasn't yet run.

```
double LastTime() const
```

Definition: Selector. Returns the time the experiment ran till, or 0 if the experiment hasn't yet run.

```
unsigned long MumID() const
```

Definition: Selector. Returns the id of the mother experiment or 0 if there was no mother.

```
unsigned long DadID() const
```

Definition: Selector. Returns the id of the father experiment or 0 if there was no father.

```
string ProcessorName() const
```

Definition: Selector. Returns the name of the machine that the experiment is running on, or null if its not running.

```
pid_t ProcessID() const
```

Definition: Selector. Returns the process ID of the process running the remote experiment. The process ID will be on the same machine that's running gacell, not the process id on the remote machine. Result is only valid if the experiment is actually running.

```
pid_t Run()
```

Definition: Runs the experiment on a remote machine. The process id is returned. If there were no processes available, the experiment is not run and 0 is returned.

```
void FinishRun()
```

Definition: Updates the experiment object once it becomes known that the process running the experiment remotely has terminated. The main things to do are to read the fitness information from fitness.id, delete that file, and to mark the processor as vacant, so that another experiment can run on it.

```
bool ShouldBeResubmitted() const
```

Definition: Returns true if the experiment aborted once before and hasn't already been resubmitted.

```
void ReinitialiseRun()
```

Definition: A reconstructor. Sets the status back to waiting, the fitness, step size, last time and pid to 0, the processor name to null, and the resubmitted flag to true.

```
unsigned long Age() const
```

Definition: Selector. Returns the number of time units this experiment has resided in the breeding population.

```
void GetOlder()
```

Definition: Modifier. Increments the time this experiment has resided in the breeding population by one.

```
void WriteExpFileInit()
```

Definition: Creates an exp.id file for experiments that are initialised using constructors other than the breeding constructor. This is the 'command' file sent to the cellsim program to run the cell simulation experiment. This file consists of the initial concentrations for the cell (read from the 'initconc' file) followed by the genome to be used.

```
void WriteExpFileBreed()
```

Definition: Creates an exp.id file for experiments that are initialised using the breeding constructor. This is the 'command' file sent to the cellsim program to run the cell simulation experiment. This file consists of the initial concentrations for the cell (taken as the final concentrations of the mother cell, unless that cell died, in which case they're read from the initconc file) followed by the genome to be used.

ProcessorPool

This class manages the allocation of all of the processors used to run experiments. Processors are booked out of the pool when an experiment is run, then returned when the run finishes. The pool shares experiments among the processors using a fair scheme. There is only one processor pool object for the program. This means that the constructors are private. Allows only one ProcessorPool to be constructed per program.

```
class ProcessorPool
{
    OVector<Processor> m_pool;
    long m_numFree; // number of free processors available.
    long m_idx; // index of the processor in m_pool to run on next.
};
```

```
static ProcessorPool& Pool()
```

Definition: Returns the one and only processor pool in the program. This is the only way to get access to a ProcessorPool. The first time Pool is called, the ProcessorPool is constructed using the 'pool' file.

```
int FreeProcessors() const
```

Definition: Returns true if there are free processors in the pool.

```
string Run()
```

Definition: Finds a free processor, books it for the run, and returns its name. An empty string is returned if there are no processors free. Select a new machine rather than use up all the runs available on the one machine.

```
void Finish(string const & processor, bool experimentAborted)
```

Definition: Release a processor back to the pool and complain if the processor caused the experiment to abort.

```
ProcessorPool(istream& in)
```

Definition: Constructor. Read the processor names and the maximum number of runs for each processor from an input stream. Usually this comes from a file called 'pool'.

Processor

This class represents a physical machine on which a number of experiments can be run.

```
class Processor
{
    string m_name; // the name of the processor
    int m_currRunning; // the number of simulations currently running
    int m_max; // the max number of simulations allowed on this
                // processor.
    int m_abortCount; // the number of consecutively aborted experiments
                    // run on this processor. If this gets too high,
                    // the processor is not allowed to run any more.
};
```

```
Processor(string const & name, int maxToRun)
```

Definition: Constructor. Creates a processor with a given name and maximum number of simultaneously running simulations.

```
string const & Name() const
```

Definition: Selector. Returns the name of the processor.

```
int MaxAllowed() const
```

Definition: Selector. Returns the maximum number of simultaneously running simulations on the machine.

```
int NumNowRunning() const
```

Definition: Selector. Returns the number of simulations currently running on the processor.

```
int CanRun() const
```

Definition: Returns 1 if the processor can run another simulation and 0 if the processor is already running its maximum number of simulations.

```
void RunASim()
```

Definition: Runs a simulation on this processor.

```
bool FinishRunning(bool experimentAborted)
```

Definition: Tell the processor that the simulation running on it has now finished and perhaps complain that the processor caused the simulation to abort. Returns true if the processor should be used for future runs and false if we should avoid using it from now on.

CellSim

Main Program

Name: cellsim

Definition: Free Program. Runs and scores a cell simulation. Reads an experiment file, builds an appropriate cell and simulates metabolism in the cell for a given time period. Scores the cell simulation and produces a fitness.id file. Reads the file cell.par for parameters common to all simulations.

Usage: cellsim [-v -f] n

where n is the experiment number.

-v means verbose mode. Produce output files cell.dsc.n and cell.out.n

-f means output data at fixed time steps. Those listed in cell.par as the time step size. Otherwise, make the largest steps possible with accuracy maintained.

Functions:

Name: main

Definition: The main line of the cellsim program. Sets up an experiment, carries it out and determines the fitness.

Name: ReadInitialWorldConcs

Definition: Reads the concentrations for chemicals in the environment and returns them in an ordered list. Only ordinary chemicals should exist outside the cell, so all other types of chemicals are ignored.

Name: ReadCellInfo

Definition: Reads information to create the cell from the exp.id file. Important information read is the initial concentrations of chemicals in the cell, the genome and the initial number of water and cell wall molecules. The latter numbers of molecules also have default values in the cell.par parameter file. Some variables are not inherited from the mother cell. These include bound carrier molecules, switch variables, and partly expressed protein variables.

Name: ShouldAbort

Definition: Determines whether the cell has reached a state where it should be aborted. This usually means cell death. Returns 1 if the cell simulation should be aborted, 0 otherwise.

Name: WriteFitness

Definition: Calculates the fitness of the cell and writes it to a fitness.id file. Returns the fitness of the cell.

Name: CalcSwitchCorrelation

Definition: Calculates and returns the correlation between the operon promoter regions and the species of membrane building and breaking chemicals.

*Environment and Cell classes***Vessel**

An abstract base class representing a container in which chemicals can react together. Provides the interface for chambers containing reacting chemicals. Holds a number of variables and differential equations for the chemicals and other variables, although the actual initialisation and use of these variables is under the control of child classes. All vessels contain a number of water molecules and a particular volume and radius (spherical shape is assumed). Vessels provide the interface for the numerical integration of the DEs to calculate the value of variables in the next time step.

```
class Vessel
{
    typedef TwoDBArr<Chemical> tVarArray;

    enum VarType    // primary index into the variable and DE arrays.
    {
        vt_ord = 0,    // an ordinary chemical variable / DE
        vt_ord_bar = 1, // the bar variable / DE for the ordinary chemical
        vt_enz = 2,    // an enzyme variable / DE
        vt_enz_bar = 3, // the bar variable / DE for an enzyme
        vt_pex = 4,    // a partly expressed enzyme variable / DE
        vt_car_e = 5,  // a carrier variable (exterior) / DE
        vt_car_i = 6,  // a carrier variable (interior) / DE
        vt_bnd_e = 7,  // a bound carrier variable (exterior) / DE
        vt_bnd_i = 8,  // a bound carrier variable (interior) / DE
        vt_switch = 9, // a gene switch variable / DE
        vt_water = 10, // the water variable / DE
        vt_wall = 11   // the cell membrane variable / DE
    };

    typedef TwoDBArr<CellDiffEqu> tDEArray;

    // Constants
    static long const kNumVarTypes; // number of variable types
    static char const * const kVarDescs[]; // their descriptions

    static TwoDPos const kWATERPos; // idx of water var/DE
    static TwoDPos const kWallPos; // idx of wall var/DE

    // Member variables ...

    double m_time; // Current time

    // The variables array ...
    tVarArray* m_vars;

    // The differential equations array ...
    tDEArray* m_DEs;

    // A dictionary to find a given chemical in the variable and DE
    // arrays.
    ODictionary<Chemical, TwoDPos> m_map;
};
```

Vessel()

Definition: Default constructor for the Vessel class.

virtual double Time() const

Definition: Selector. Returns the current simulation time.

```
virtual double Water() const
```

Definition: Selector. Returns the number of water molecules in the Vessel.

```
virtual long NumDEs() const
```

Definition: Returns the number of differential equations in the array.

```
virtual double OneMoleculeThreshold() const
```

Definition: Calculates and returns the concentration of a chemical species comprising one molecule in the vessel. This is the inverse of the number of water molecules in the cell.

```
virtual double Volume() const
```

Definition: Calculates and returns the volume of the vessel based on the number of water molecules it contains. A plump sphere is assumed. The volume is calculated using the formula: $V = (AW/NA) * 1.0e-6 * NW$, where AW is the atomic mass of water = 18.0152, NA is Avogadro's number = $6.0220e+23$, and NW is the number of water molecules in the cell

```
virtual double Radius() const
```

Definition: Calculates and returns the current radius of the Vessel given its current volume. The radius is calculated using the formula: $R = (3V / 4\pi) ** 1/3$, where V is the volume and pi is 3.14159265.

```
virtual double CalcNextValues(double interval, bool
doLargestPossibleStep = false) = 0
```

Definition: A pure virtual function giving the interface of a function that calculates the next set of values for the variables at the current time plus an interval. If an argument doLargestPossibleStep is true, the interval is ignored and the values are calculated for the largest step possible keeping a reasonable error rate. The actual time step taken is returned.

```
static TwoDPos const & WaterPos()
```

Definition: Selector. Returns the index of the variable for the number of water molecules in the variable array.

```
static TwoDPos const & WallPos()
```

Definition: Selector. Returns the index of the variable for the number of cell membrane molecules in the arrays.

```
virtual OVector<Vessel::ChemOut> Concs() const
```

Definition: Returns a list of the variables in the Vessel. We can't simply return a vector of Chemical because two variables can have the same Chemical name, but refer to two different variables. For example, an enzyme and its bound form. This function simply looks through the array and forms a ChemOut which is the pair of row name (ie, its VarType) and the Chemical.

```
virtual void Time(double)
```

Definition: Modifier for children only. The only way to modify the current time in the Vessel.

```
virtual tVarArray& Vars()
```

Definition: Modifier. Returns a reference to the variables array allowing children to modify variables.

```
virtual tDEArray& DEs()
```

Definition: Modifier for children only. Returns a reference to the DEs array allowing modification of differential equations.

```
virtual ODictionary<Chemical, TwoDPos>& Map()
```

Definition: Modifier for children only. Returns a reference to the map in the Vessel allowing modification. (The map associates chemicals with their position in the arrays).

```
virtual tVarArray const & Vars() const
```

Definition: Selector. Overloading Vars above, but a const version that is read only.

```
virtual tDEArray const & DEs() const
```

Definition: Selector. Overloading DEs above, but a const version that is read only.

```
virtual ODictionary<Chemical, TwoDPos> const & Map() const
```

Definition: Selector. Overloading Map above, but a const version for read only access only.

```
void ConstructArrays()
```

Definition: This function allocates the storage for the variables and differential equations arrays. It is the child's responsibility to call this when it's ready to do the construction and to initialise the values in the arrays.

ChemOut

A really simple class declared publicly inside Vessel represents a variable in the Vessel. Chemical isn't enough because two variables may have the same chemical. The combination VarType and Chemical is needed.

```
struct ChemOut
{
    VarType m_vt;
    Chemical m_chem;
};
```

```
ChemOut(VarType vt, Chemical const & chem)
```

Definition: Constructor.

World

Represents the environment of a cell which also contains the cell itself. Derived publicly from Vessel. Initialises the chemicals in the environment and the cell. Calculates the state of the variables and the state of the cell for time steps.

```
class World : public Vessel
{
    Cell m_cell;
};
```

```
World(OOrder<Chemical> const & worldInitial, OOrder<Vessel::ChemOut>
const & cellInitial, double cellNumWater, double cellNumWall, bool
verbose)
```

Definition: Constructor. Initialises the world and a cell with a random genome. Most of the initialisation is done by calling the Init method.

```
World(OOrder<Chemical> const & worldInitial, OOrder<Vessel::ChemOut>
const & cellInitial, CellChromo const &, double cellNumWater, double
cellNumWall, bool verbose)
```

Definition: Constructor. Initialises the world and cell. This constructor requires a genome for the cell. Most of the initialisation is done by calling the Init method.

```
Cell const & TheCell() const
```

Definition: Selector to view the cell in the environment.

```
double ConcOf(Chemical const &) const
```

Definition: Returns the concentration of the requested chemical in the environment. Uses the map to find the index and returns the Amount of the Chemical. If the chemical isn't in the map, we can assume that it has zero concentration.

```
virtual double CalcNextValues(double interval, bool
doLargestPossibleStep = false)
```

Definition: The implementation of the method from Vessel to calculate the value of variables at the time $t + \text{interval}$, or using the largest step possible with a given error rate. Since the variables in the environment are fixed, just send the cell the CalcNextValues message and update the step.

```
void Init(OOrder<Chemical> const &)
```

Definition: Initialises the chemicals in the world.

```
void InitValues(OOrder<Chemical> const & ordChems, OOrder<Chemical>
const & initial)
```

Definition: Initialises the values in the variable array to given values.

Cell

This class represents a cell containing a genome and metabolism encoded in a system of differential equations which can be numerically integrated over a number of time steps. It is derived publicly from Vessel. Constructs experiments with a random genome for the cell, breeds from two parent cells, performs cell simulations.

```
class Cell : public Vessel
{
    bool m_verbose; // true: output info to file, otherwise be silent

    World* m_pWorld; // a link to the world the cell lives in

    // The genome.
    CellChromo m_genome; // the raw genome
    OVector<ParsedOperon> m_ParsedOperons; // the genome parsed into
enzymes

    OOrder<Chemical> m_carriers; // the cell's carrier molecules
    OOrder<Chemical> m_spiders; // molecules the cell can use for
// transcription
    OOrder<Chemical> m_buildCellWall; // chemicals that can be used to
// build the cell wall
    OOrder<Chemical> m_breakCellWall; // chemicals that can be used to
// break the cell wall
};
```

```
Cell(World* pWorld, OOrder<Vessel::ChemOut> const & initial, double
initWater, double initWall, bool verbose)
```

Definition: Constructor. This constructor creates a random genome. Most of the work is done by calling the Init function.

```
Cell(World* pWorld, OOrder<Vessel::ChemOut> const & initial,
CellChromo const &, double initWater, double initWall, bool verbose)
```

Definition: Constructor. This constructor takes a genome for the cell. Again, most of the work is done using the Init method.

```
CellChromo const & Chromosome() const
```

Definition: Selector. Returns the raw genome used in the cell.

```
OVector<ParsedOperon> const & ParsedOperons() const
```

Definition: Selector. Returns a reference to the list of parsed genes in the cell.

```
double VolumeNoAction(double time, OVector<Vessel::ChemOut> const &
initialConcs) const
```

Definition: Estimates and returns the volume of the cell treating the cell as if it has taken no metabolic, genetic or facilitated diffusion action. Only background diffusion is assumed to have occurred. The volume is calculated for the given time and using initialConcs as the starting concentrations of the chemicals. This function is used to calculate the fitness of the cell.

```
virtual double CalcNextValues(double interval, bool
doLargestPossibleStep = false)
```

Definition: Calculate the next set of values for the variables in the differential equations using a process of numerical integration. The actual time step made is returned. See also CalcNextValues in the Vessel class.

```
OOrder<Chemical> const & Spiders() const
```

Definition: Selector. Returns an ordered list of chemical species able to be used for gene expression.

```
OOrder<Chemical> const & Builders() const
```

Definition: Selector. Returns an ordered list of chemical species able to build up the cell membrane.

```
OOrder<Chemical> const & Breakers() const
```

Definition: Selector. Returns an ordered list of species able to break down the cell membrane.

```
long NumPartials() const
```

Definition: Selector. Returns the number of partially expressed proteins with at least one molecule in the cell.

```
void Init(OOrder<Vessel::ChemOut> const &, double, double)
```

Definition: Initialises the cell, by parsing the genome and setting up the differential equations

```
void ParseGenome()
```

Definition: Parses the raw genome to create a list of 'parse trees' – one for each operon.

```
OOrder<Chemical> GetCarrierList()
```

Definition: Returns a list of all carriers mentioned in the genome.

```
void AddMonomers(OOrder<Chemical>&)
```

Definition: Adds a chemical representing the chemical species for a monomer (o0-o9) to the ordered list passed in as an argument.

```
void InitArrays(OOrder<Chemical> const & ordinaryChemicals,
OOrder<Chemical> const & enzymes, OOrder<Vessel::ChemOut> const &,
double, double)
```

Definition: Initialises each row of the variables and differential equations arrays.

```
void InitOrdChems(OOrder<Chemical> const & ordinaryChemicals,
OOrder<Vessel::ChemOut> const &)
```

Definition: Initialises the ordinary chemicals variables and differential equations.

```
void InitEnzChems(OOrder<Chemical> const & enzymes,
OOrder<Vessel::ChemOut> const &)
```

Definition: Initialise the enzyme and bar enzyme rows of the variable and differential equations arrays. The algorithm is basically the same as the one for InitOrdChems except that if an enzyme variable has no initial value, we initialise it to half of the one molecule threshold concentration instead of the much smaller 1.0e-25.

```
void InitCarChems(OOrder<Vessel::ChemOut> const &)
```

Definition: Initialises the internal and external carrier rows of the variable and differential equation arrays. The algorithm is the same as InitOrdChems except that if an internal carrier variable has no initial value, it's set to half of the one molecule threshold concentration, like the enzymes.

```
void InitSwitches(OOrder<Chemical> const & ordinaryChemicals,
OOrder<Chemical> const & enzymes, OOrder<Chemical> const &
allChemicals)
```

Definition: Initialises the variables representing the degree to which each operon is switched on.

```
void ConsiderForSwitching(Chemical const &, long operonNumber, TwoDPos
const & operonIndex)
```


Definition: A simple helper routine for InitSwitches that checks to see if a candidate chemical can be used to switch the given operon. If it can the DE is changed accordingly.

```
void InitWall(OOrder<Chemical> const & ordinaryChemicals,
OOrder<Chemical> const & enzymes, double)
```

Definition: Initialises the number of cell membrane molecules variable and differential equation.

```
void InitWater(OOrder<Chemical> const & ordinaryChemicals,
OOrder<Chemical> const & enzymes, double initialNumber)
```

Definition: Initialises the number of water molecules variable and differential equation. This variable and DE aren't strictly necessary because they can be calculated from the number of wall molecules variable, but they exist for historical reasons. Since the number of water molecules depends on the number of cell wall molecules, we can't guarantee that we will initialise the number of water molecules to the value passed in. This value is used solely as a reference for the concentration of the ordinary chemicals for use in the differential equation.

```
void InitOrdinaryReactions()
```

Definition: Adds terms to the differential equations involved in each of the metabolic reactions allowable in the cell.

```
void InitTranscriptionReactions(OOrder<Chemical> const &
ordinaryChemicals)
```

Definition: Adds terms to existing differential equations and adds new variables and differential equations (for partially expressed proteins) to allow operon expression and protein production. Most of this work, however, is deferred to another method. This method simply identifies all the spiders from the list of chemicals passed in, then calls another method to do the rest.

```
void InitDegradationReactions(OOrder<Chemical> const &
chemicalsToDegrade)
```

Definition: Determines the reactions necessary to degrade the chemicals (usually enzymes and carriers) into their constituent parts. These reactions are then applied to the differential equations affected to add the appropriate terms.

```
void HandleTransReactions(OOrder<Chemical> const & spiders)
```

Definition: Given a list of chemical species that can act as spiders, this method gets a list of the reactions that will eventuate from the expressing the operons. These reactions are then sent to the differential equations concerned for consideration. The DEs may add a term if applicable.

```
OVector<TransReaction> ReadTransReacts(OOrder<Chemical> const &
spiders, Chemical[] monomerSpecies, TwoDPos[] monomerIndices)
```

Definition: Makes and returns a list of the transcription reactions required given the genome for each of the spiders. Producing a protein may need a number of intermediary chemical species that act as partially built proteins bound to a spider. These chemical species are allocated the names p0, p1, etc. This method also determines what partially built proteins are needed and allocates a variable and differential equation for each. The value of each variable and the terms for the differential equation are determined later.

```
OVector<TransReaction> GetReactions(TwoDPos const & operonIdx,
ParsedOperon const & operon, int pExpBase, Chemical const & spider,
TwoDPos const & spiderIdx, Chemical[] monomers, TwoDPos[] monomerIdxs)
```

Definition: Returns a list of reactions necessary to transcribe/translate the enzymes and carriers on the given operon using the given spider. Partly expressed protein variables will start being numbered from pExpBase. Note: We need to know what the first and last chemicals encoded on the operon are. The first base for the first chemical on an operon is switchable and needs to be handled specially. The last base on the last gene needs to release the spider back into the 'spider pool'.

```
OVector<TransReaction> GetReactionsFor(Chemical const & chem,
OVector<int> const & baseList, bool firstChemical, bool lastChemical,
TwoDPos const &, int& p, Chemical const & spider, TwoDPos const &
spiderIdx, Chemical[] monomers, TwoDPos[] monomerIdxs)
```

Definition: Returns a list of reactions necessary to transcribe/translate the given chemical/baseList on the given gene using the given spider. Partly expressed protein variables will start being numbered from the

value `p` passed in. `p` is changed by this method. `firstChemical` should be true if `chem` is the first chemical on the gene. Similarly `lastChemical` should be true if `chem` is the last chemical in the gene.

```
void InitDiffusionReactions(OOrder<Chemical> const & carriers)
```

Definition: Initialises the carrier facilitated diffusion reactions, adds variables and DEs and adds terms to other applicable DEs.

```
void ResetVariables(tVarArray& vars, tVarArray const & kOrigVars)
```

Definition: Resets the values in `vars` to the values in `kOrigVars`, or 0 if the value wasn't in `vars`.

```
void ReallocateArrays()
```

Definition: Reallocates the temporary arrays used in `CalcNextValues`. Basically this simply involves deleting the storage for them and constructing new ones copied from `Vars()`. Then the `m_arraysOutOfDate` flag is set to false.

```
void Derivs(double t, tVarArray const & y, tVarArray& dydt)
```

Definition: Calculates the derivatives at time `t` for `y`. Stored in `dydt`. This function is used for the numerical integration. Since our differential equations are autonomous, the time `t` is not used.

```
void RungeKutta4(tVarArray const &, tVarArray&, double, double, tVarArray&)
```

Definition: The lowest level numerical integration function called by drivers. This function performs a fourth-order Runge-Kutta numerical integration. This function comes from Press et al. (1995). One modification made to this algorithm is that we never should get negative values. Any values calculated to be less than 0 here, are set to 0.

```
void RKQC(tVarArray&, tVarArray&, double&, double, double, tVarArray&, double&, double&)
```

Definition: A fifth-order Runge-Kutta numerical integration function that adapts the step size to keep the local truncation error within a given accuracy. This function calls `RungeKutta4` to make the actual step. This function too comes from Press et al. (1995).

```
void ODEInt(tVarArray const &, double, double, double, double, double, tVarArray&, int&, int&)
```

Definition: A Runge-Kutta driver with adaptive step size control. This algorithm comes from Press et al. (1995).

```
double ODEIntLargest(tVarArray const &, double, double, double, double, tVarArray&, int&, int&)
```

Definition: This function is the same as `ODEInt` above except for a few important differences. It always makes the largest step possible and remembers the last time step it made. This value is saved so that the time step estimate can be retained between calls to `ODEIntLargest`. There is no high bound for integration. The routine just makes one step: the largest possible.

```
OOrder<Chemical> FindNewChemicals(tVarArray const & newV)
```

Definition: Returns a list of chemical species whose concentrations have increased over the one molecule threshold for the first time. Only ordinary chemicals and enzymes need be checked (`newV`). The purpose of finding the new chemicals is to relay this information to the Chemistry in order to determine the new reactions that are possible. Only ordinary chemicals and enzymes will generate more metabolic reactions.

```
void UpdateOldVariables(tVarArray const & newV, double oldNW)
```

Definition: Copies the new values of the variables (`newV`) to the standard set of variables (`m_vars`). If the number of water molecules has changed, a correction factor is applied to all concentrations (NOT gene switches or the number of water or wall molecules) by calling `CorrectConcentrations`.

```
bool CorrectConcentrations(double oldNW, double newNW)
```

Definition: If the before and after number of water molecules values are different, a correction factor is calculated and applied to all concentration variables in the `m_vars` array. If the values are the same, no change is made to the variables. Returns true if concentrations were corrected, false otherwise. The correction factor is applied because concentrations are defined in terms of the number of water molecules

in the cell, so if this value changes, so does the concentration (to keep the number of molecules of x constant).

```
bool HandleNewChems(OOrder<Chemical> const & newChems)
```

Definition: Looks for newly produced chemicals and chemicals that will need to be added to the cell's list of variables and DEs. If there are any, variables and DEs are created for them and their effect on other variables through the DEs are computed. Returns true if the changes were made, false if no new chemicals were found.

Genome classes

Haploid

An abstract base class representing the interface required by all haploids. Haploids are components of genetic material for a member of the population. Acts as a base class for all kinds of haploids, giving the specification of exactly what behaviour is required for a haploid.

```
class Haploid
{
    // No member variables
};
```

```
Haploid()
```

Definition: Default constructor constructs an empty haploid.

```
Haploid(ORNG const &)
```

Definition: Constructor constructs a random haploid.

```
virtual Haploid* newClone() const = 0
```

Definition: Virtual copy constructor, returns a copy of the actual object as a newed object. To be implemented by children.

```
virtual void Crossover(Haploid&) = 0
```

Definition: Performs recombination between two haploids. Both are changed. To be implemented in children.

```
virtual void Invert() = 0
```

Definition: Performs the inversion genetic operator on this haploid. To be implemented in child classes.

```
virtual void Mutate() = 0
```

Definition: Mutates the haploid. To be implemented in children.

Chromosome

An abstract base class representing a chromosome: the genetic material of a member of the population of cells. Acts as a template for all chromosomes in the system, giving all methods necessary for all chromosomes.

```
class Chromosome
{
    Haploid* m_pLeft;    // left haploid (NOT strand)
    Haploid* m_pRight;  // right haploid (NOT strand)
};
```

```
Chromosome()
```

Definition: Default constructor constructs an empty chromosome (ie. with no haploids).

```
Chromosome(ORNG const &)
```

Definition: Constructor constructs a random chromosome. It is the child's responsibility to make sure the haploids are random since this Chromosome doesn't know what kinds of Haploid are being stored.

```
Chromosome(Chromosome const & mum, Chromosome const & dad)
```

Definition: Breeding constructor creates a chromosome by breeding two parents together. There are two cases to consider:

If we're working with single haploid chromosomes, then: mumHaploid, dadHaploid → cross over, invert and mutate crossedMumHaploid, crossedDadHaploid → choose one at random for the child's haploid.

If we're working with double haploid chromosomes, then: mumLeft, mumRight, dadLeft, dadRight → cross over, invert and mutate each parent; crossMumLeft, crossMumRight, crossDadLeft, crossDadRight → choose a random mum and dad haploid; randomMum, randomDad → choose one of these at random for the left and right child's haploid

The child class need take no other action.

```
Haploid* TakeLeft(Haploid* pLeft)
```

Definition: Initialises the chromosome's left haploid. This should be used by the child chromosome class constructors (not the breeding one though). The previous left haploid is returned. Once the haploid is given to the chromosome, it will take care of the pointer.

```
Haploid* TakeRight(Haploid* pRight)
```

Definition: Exactly the same as TakeLeft, but for the right haploid.

```
bool HasLeft() const
```

Definition: Returns 1 if the chromosome has a left haploid, 0 otherwise.

```
bool HasRight() const
```

Definition: Same as HasLeft, but for the right side.

```
Haploid const * Left() const
```

Definition: Returns a read only version of the leftmost haploid or 0 if there is no left haploid.

```
Haploid* Left()
```

Definition: Overloaded version of the previous method that returns the haploid itself in a writeable form.

```
Haploid const * Right() const
```

Definition: A similar method to the read-only Left(), above.

```
Haploid* Right()
```

Definition: A similar method to the writeable Left(), above.

```
void SingleHaploidBreeding(Haploid* pMum, Haploid* pDad)
```

Definition: Performs the steps necessary to make a child haploid from breeding two parent haploids and puts the result into the Chromosome.

```
void DoubleHaploidBreeding(Haploid* pMumLeft, Haploid* pMumRight,
Haploid* pDadLeft, Haploid* pDadRight)
```

Definition: Performs the steps necessary to breed a child chromosome from two parent chromosomes when the chromosomes are diploid.

NibbleMixin

A simple abstract base class that is 'mixed in' to child classes with multiple inheritance to provide nibble functionality - ie, be able to read the haploid in 4 bit blocks. Provides the interface for methods needed to ensure access to a haploid in 4 bit blocks.

```
struct NibbleMixin
{
```

```
// No member variables
};
```

```
virtual unsigned int Length() const = 0
```

Definition: Returns the number of nibbles.

```
virtual bool IsEmpty() const = 0
```

Definition: Returns 1 if there are no nibbles (ie, Length() == 0), or 0 otherwise.

```
virtual OBitVec operator[](unsigned int nibblei) const = 0
```

Definition: Returns the ith nibble in the list as a bit vector of 4 bits.

BitHaploid

Represents a haploid which stores its genetic information as a raw bit string. It's up to child classes to make sense of the bit string. Derived publicly from Haploid. BitHaploid is a specialisation of a haploid that stores the information in a bit string. It needs to be able to create a random string of any length and provide limited read access to the bits. It also needs to provide an implementation of the cross over, inversion and mutation genetic operators and allow the bit string to be read from an input stream.

```
class BitHaploid : public Haploid
{
    OBitVec bits;
};
```

```
BitHaploid(unsigned int length)
```

Definition: Constructor constructs an empty haploid.

```
BitHaploid(unsigned int length, ORNG const &)
```

Definition: Constructor - creates a random haploid of a particular length.

```
BitHaploid(istream& in)
```

Definition: Constructor - creates a haploid by reading the bits out of an input stream. 0 and 1 characters are expected. The first non 0 or 1 marks the end of the bit string. This character is replaced back into the stream.

```
virtual void Crossover(Haploid&)
```

Definition: Performs recombination between two bit haploids. Both haploids must be bit haploids (or their children).

```
virtual void Invert()
```

Definition: Possibly performs an inversion operation on the bit string making up the BitHaploid. If the actual inversion is to occur, it involves choosing two points on the haploid and reversing the bit string contained therein. To determine whether the inversion should occur, we choose a random number in [0, 1]. If this number is less than or equal to the inversion rate (as read from the parameters in cell.par) the inversion will occur.

```
virtual void Mutate()
```

Definition: Mutates the genome.

```
OBitVec const & Bits() const
```

Definition: Selector. Returns the bit string.

```
OBitVec& Bits()
```

Definition: Overloaded version of the previous function that allows write access to the bit string for children.

DoubleStrandBitHaploid

This is a haploid that stores its genetic information as a raw bit string. It uses two strands - the second is the reversed complement of the first strand. The main change is in the inversion operation. It is publicly derived from BitHaploid. The main responsibility of this class is that it has to keep the second strand updated with the first strand. We store the second strand as a separate object, rather than computing from the one strand, to make the program faster.

```
class DoubleStrandBitHaploid : public BitHaploid
{
    OBitVec secondStrand;        // the computed strand
};
```

```
DoubleStrandBitHaploid(unsigned int length)
```

Definition: Constructor makes an empty double stranded bit haploid of the requested length. The length here is the length of one strand. There will be twice this number of bits in the genome.

```
DoubleStrandBitHaploid(unsigned int length, ORNG const &)
```

Definition: Constructor creates a random double stranded bit haploid. Again the length is the length of one strand. The second strand will be the ones complement of the first strand.

```
DoubleStrandBitHaploid(istream& in)
```

Definition: Constructor creates a double strand bit haploid by reading it from an input stream. The constructor expects the bits for a first strand, followed by a '*', and the bits for the second strand. The constructor ensures that the second strand is the reversed ones complement of the first strand by ignoring all bits in the second strand and calculating them from the first strand anew.

```
virtual void Crossover(Haploid&)
```

Definition: Recombination of haploids. Simply calls the parent Crossover, then calls UpdateSecondStrand() to make sure the strands are synchronised.

```
virtual void Invert()
```

Definition: Performs an inversion operation on the haploid. Start and end points on the first strand are selected and the bits in between are negated and their order reversed. UpdateSecondStrand() is called to synchronise the strands. This operation effectively moves swaps the regions of the two strands.

```
virtual void Mutate()
```

Definition: Mutates the haploid. The routine simply calls the parent's Mutate method, then synchronises the strands with a call to UpdateSecondStrand().

```
OBitVec const & Strand1() const
```

Definition: Selector. Returns the bit string for the first strand.

```
OBitVec const & Strand2() const
```

Definition: Selector. Returns the bit string for the second strand.

```
void UpdateSecondStrand()
```

Definition: Synchronises the second strand to be the complement of the first strand.

DoubleStrandNibbleHaploid

This class is the same as a DoubleStrandBitHaploid, except that instead of reading the haploid at a bit granularity, we read it in nibbles (blocks of 4 bits). It is publicly derived from DoubleStrandBitHaploid and publicly from NibbleMixin. The main responsibility that this class has is to provide the nibble blocking of the genome. It ensures that crossover and inversion only occurs on nibble boundaries and Gray codes the nibbles if required (which usually isn't the case).

```
class DoubleStrandNibbleHaploid : public DoubleStrandBitHaploid,
public NibbleMixin
{
    // No member variables
};
```

```
DoubleStrandNibbleHaploid(unsigned int nNibbles)
```

Definition: Constructor creates a new empty double strand nibble haploid containing nNibbles in each strand.

```
DoubleStrandNibbleHaploid(unsigned int nNibbles, ORNG const &)
```

Definition: Constructor creates a random double strand nibble haploid containing nNibbles in each strand.

```
DoubleStrandNibbleHaploid(istream& in)
```

Definition: Creates a new double strand nibble haploid by reading it from an input stream.

```
virtual unsigned int Length() const
```

Definition: Returns the number of nibbles in the genome. ie, the number of nibbles in each strand added together.

```
virtual bool IsEmpty() const
```

Definition: Returns true if there are no nibbles in the genome.

```
virtual OBitVec operator[](unsigned int nibblei) const
```

Definition: Returns the ith nibble on the genome. i=0 is the first nibble of the first strand and i=n is the first nibble of the second strand. Remember that reading of the strands goes forwards through the first strand and backwards through the second strand. If the cell.par parameter file states that Gray coding required the nibbles will be applied. A bit vector containing four bits is returned.

```
virtual void Crossover(Haploid&)
```

Definition: Crossover is similar to the parent class except that it only occurs on nibble boundaries.

```
virtual void Invert()
```

Definition: Inversion is the same as for the parent except that it only occurs on nibble boundaries.

CellChromo

This class represents the chromosome for a cell. It is publicly derived from Chromosome. The responsibilities of a cell chromosome are construction, reading of bits and parsing into parsed genes, and genetic operators.

```
class CellChromo : public Chromosome
{
    OVector<ParsedOperon> m_ops; // the operons we've decoded so far
    ParsedOperon m_currOp;      // the operon we're currently reading
    OVector<int> m_currEnz;     // the enzyme we're currently reading
    OVector<int> m_currCarrier; // the carrier we're currently reading
    OVector<int> m_currPromoter; // the promoter we're currently reading
};
```

```
CellChromo()
```

Definition: Constructor creates an empty chromosome. Calls TakeLeft passing in a new DoubleStrandNibbleHaploid and TakeRight passing in 0. The left hand haploid's length is read from the parameter file cell.par.

```
CellChromo(ORNG const &)
```

Definition: Constructor creates a random cell chromosome. Calls TakeLeft passing in a new random DoubleStrandNibbleHaploid and TakeRight with 0. Again, the number of nibbles in each strand of the left hand haploid is read from the cell.par parameters file.

```
CellChromo(CellChromo const & mum, CellChromo const & dad)
```

Definition: Breeding constructor creates a new cell chromosome from the chromosomes of two parent cells.

```
CellChromo(istream& in)
```

Definition: Constructor creates a chromosome by reading it from an input stream.

```
virtual OVector<ParsedOperon> ReadStrip()
```

Definition: Decodes the genome and builds a parsed list of the operons. A list of ParsedOperon is returned.

The following methods are a part of the parsing algorithm:

```
String changeToNotInOperon(int base)
```

Definition: We must have just finished an operon. The change from function will clean up and save the gene, so we don't need to do anything here.

```
String stayInNotInOperon(int)
```

Definition: We're still in a non-coding region of the genome, so we don't need to do anything here.

```
String changedFromNotInOperon(int)
```

Definition: We're starting an operon, but the work initialising it will be done later.

```
String changeToStartOfOperon(int)
```

Definition: We've just read the *<start operon>* base. There's nothing to do as yet.

```
String stayInStartOfOperon(int)
```

Definition: We had read a *<start operon>* base and we've just read another one. Forget about the last aborted operon and concentrate on the new one.

```
String changedFromStartOfOperon(int)
```

Definition: We had just read a *<start operon>* and now we're either starting a gene or going back to a non coding region, but we'll leave the work up to the 'changed to' function.

```
String changeToInPromoter(int)
```

Definition: We had just read a *<start operon>* and now we just read a monomer. That means we're in a promoter region. If this first base in the promoter region is even then this is a repressible operon which is on until turned off. If the base is odd, then it's an inducible operon.

```
String stayInInPromoter(int)
```

Definition: We were in the promoter region and we just read another monomer so we're still in the promoter region. Just add this monomer to the promoter string we're accumulating (*m_currPromoter*).

```
String changedFromInPromoter(int)
```

Definition: We were in the promoter region but we've read a special code that marks the end of the promoter region. One of two things can happen here. If we just read a *<start enzyme>* or *<start carrier>* code then we're going to start a gene. This means we need to store the now complete promoter in the operon structure. If, on the other hand, we read a *<start operon>* or an *<end operon>* the operon we were in (which had no genes) is of no use and should be aborted.

```
String changeToStartOfEnzyme(int)
```

Definition: We've just read the *<start enzyme>* code, so now we're in an enzyme. There's no work to do yet.

```
String stayInStartOfEnzyme(int)
```

Definition: We had just read a *<start enzyme>* and now we've just read another. Forget about the other gene that didn't go anywhere and think about this new one.

```
String changedFromStartOfEnzyme(int)
```

Definition: We had read a *<start enzyme>* and now one of three things has happened. We may have read a monomer in which case we'll start off the enzyme properly in the 'changed to' function. We may have read a *<start carrier>* symbol in which case we'll forget about this non-existent enzyme and start a carrier. The last option is that we just read a *<start operon>* or *<end operon>* in which case we need to finish of the operon.

```
String changeToInEnzyme(int)
```


Definition: We had just read a *<start enzyme>* and now we read a monomer, so we're in the enzyme proper. These bases will form the template for the enzyme's shape. Add the code to the end of the list forming the shape of the current enzyme (`m_currEnz`).

```
String stayInInEnzyme(int)
```

Definition: We're still reading monomers in the enzyme. Just add it to the end of `m_currEnz`.

```
String changedFromInEnzyme(int)
```

Definition: We were accumulating the codes for a enzyme's shape but we've just read a special code so this is the end of the enzyme. If the code we read was *<start operon>* or *<end operon>* we need to save the operon information we've been accumulating.

```
String changeToStartOfCarrier(int)
```

Definition: We just read a *<start carrier>* symbol, so now we need to accumulate the bases describing the shape of a carrier.

```
String stayInStartOfCarrier(int)
```

Definition: We read a *<start carrier>* symbol before and now we have just read another one. Nothing needs to be done here.

```
String changedFromStartOfCarrier(int)
```

Definition: We read a *<start carrier>* last time. Now we are either starting the shape of the carrier with a monomer, we're starting an enzyme, or we're ending the operon with an *<end operon>* or *<start operon>* code. Only the ending of an operon needs to be dealt with.

```
String changeToInCarrier(int)
```

Definition: We read a *<start carrier>* last time and a monomer this time. Add the code to the current carrier shape list (`m_currCarrier`).

```
String stayInInCarrier(int)
```

Definition: We're building the carrier's shape and here's another monomer to add to the list. Add code to `m_currCarrier`.

```
String changedFromInCarrier(int)
```

Definition: We were building a carrier's shape, but we've just read a special code which will now stop the carrier. Carriers are allowed to contain only one base, so we don't have the problem we had with the enzymes.

ParsedOperon

This class represents a parsed version of an operon including the promoter region, switching characteristics, and resulting enzymes and carriers. The main responsibility of `ParsedOperon` is to keep track of the above mentioned information related to an operon.

```
class ParsedOperon
{
    enum SwitchStyle { ss_alwaysOn, ss_defaultOn, ss_defaultOff };

    enum ParsedChemType { pct_enzyme, pct_carrier };

    struct ParsedChem
    {
        ParsedChemType m_type;
        OVector<int>    m_bases;
    };

    SwitchStyle m_sstyle;    // default transcription status
    OVector<int> m_scode;    // the code to switch the gene on or off
    OVector<ParsedChem> m_items; // a combined list of the chemicals
}
```

```

    OVector< OVector<int> > m_enzs;      // a list of the enzymes
    OVector< OVector<int> > m_carriers; // a list of the carriers
};

```

```

ParsedOperon()

```

Definition: Constructor.

```

SwitchStyle GetSwitchStyle() const

```

Definition: Selector returns the type of switching allowed for this operon.

```

OVector<int> const & GetSwitchCode() const

```

Definition: Selector returns the switching string for the operon.

```

oindex NumberOfItems() const

```

Definition: Returns the number of proteins able to be produced from the operon.

```

oindex NumberOfEnzymes() const

```

Definition: Selector returns the number of enzymes able to be produced from the operon.

```

oindex NumberOfCarriers() const

```

Definition: Selector returns the number of carriers able to be produced from the operon.

```

bool IsAnEnzyme(oindex idx) const

```

```

bool IsACarrier(oindex idx) const

```

Definition: These functions return true if the protein at index idx is an enzyme or carrier respectively.

```

OVector<int> const & GetItem(oindex idx) const

```

Definition: Returns the string of bases representing the shape of the protein at index idx on the genome.

```

OVector<int> const & GetEnzyme(oindex idx) const

```

Definition: Returns the string of bases representing the shape of the idx'th enzyme in the list of enzymes.

```

OVector<int> const & GetCarrier(oindex idx) const

```

Definition: Returns the string of bases representing the shape of the idx'th carrier in the list of carriers.

```

void SetSwitchStyle(SwitchStyle)

```

Definition: Sets the type of switching used for the operon.

```

void SetSwitchCode(OVector<int> const &)

```

Definition: Sets the switching code for the operon.

```

void AddEnzyme(OVector<int> const &)

```

```

void AddCarrier(OVector<int> const &)

```

Definition: Adds an enzyme or carrier respectively to the operon with a shape determined by the supplied list of bases.

Chemicals and Chemistry classes

Chemical

This class represents a variable in the cell simulation. Usually this is a chemical, hence the name of the class, but not always. This is historical. Our initial design didn't require numbers of molecules or switch variables. Each Chemical has a type, shape and concentration. To make things a little more complicated, we have used a letter/envelope idiom to represent chemicals to minimise the overhead caused by construction and copy construction. This Chemical class is the envelope part of the pair. The letter part is polymorphically through the AbstractChemicalRep abstract base class.

Almost all the functions are delegated to the AbstractChemicalRep.

The responsibilities of this class are to control construction and destruction of the letter `AbstractChemicalRep` and to delegate all other functions to the `AbstractChemicalRep`.

```
class Chemical
{
    enum ChemType { ct_chemical, ct_enzyme, ct_partlyExp, ct_carrier,
ct_bound, ct_geneSwitch, ct_water, ct_wall };

    AbstractChemicalRep* m_rep;
};
```

`Chemical()`

Definition: Default constructor sets the rep pointer to 0.

```
Chemical(ChemDigits const & digits, int length, double amount = 0.0,
ChemType = ct_chemical, long maxCount = 0)
```

Definition: Main constructor. `digits` is the monomers in the chemical, `length` is the number of them, `amount` is the concentration, `ChemType` is the type of chemical and `maxCount` is the maximum number of chemicals allowed to point to this rep.

```
Chemical(Chemical const &)
```

Definition: Copy constructor. The basic idea here is to end up with both `Chemicals` pointing to the same `AbstractChemicalRep`, with the rep knowing that there's another envelope pointing to it. If the rep says that it already has its maximum number of envelopes then we need to clone a new rep.

```
Chemical const & operator=(Chemical const &)
```

Definition: Assignment operator.

```
~Chemical()
```

Definition: Destructor.

```
void Amount(double)
```

Definition: Change the chemical's concentration. This is delegated to the rep.

```
void ForkRep(long maxCount)
```

Definition: Creates a new rep for the `Chemical`. The new rep will be the same as the old one (but a different object). The maximum number of envelopes for this rep is set by the `maxCount` argument. This method is important because it is required when you want to make a copy of a `Chemical` and change the concentration of one of the chemicals independently.

```
string Name() const
```

Definition: Selector returns the name of the chemical. This will include the shape too. This class just delegates the work to the rep.

```
double Amount() const
```

Definition: Returns the concentration of the chemical. This work is just delegated to the rep.

```
ChemDigits const & Digits() const
```

Definition: Returns the shape of the chemical. This is just delegated to the rep. `ChemDigits` is just a typedef for a fixed size array.

```
int Length() const
```

Definition: Returns the number of digits used for the shape. (Delegated to the rep).

```
bool IsAMonomer() const
```

Definition: Returns true if the `Length()` is 1.

```
bool CanCatalyseReactions() const
```

Definition: Returns true if the chemical can catalyse reactions ie, it's an enzymes. (Delegated to rep).

```
bool CanTakePartInReactions() const
```

Definition: Returns true if the chemical can take part in reactions as a substrate molecule or product but not if it just acts as a catalyst. (Delegated to rep)

```
bool AffectsCellWall() const
```

Definition: Returns true if this chemical can build up or break down the cell wall. (Delegated to rep)

```
bool BuildsCellWall() const
```

Definition: Returns true if this chemical builds up the cell wall. (Delegated to rep)

```
bool BreaksCellWall() const
```

Definition: Returns true if this chemical breaks down the cell wall. (Delegated to rep)

```
double TranscriptionRate() const
```

Definition: Returns the transcription rate constant that would be used if this chemical were to act as a spider. (Delegated to rep)

```
double SwitchEfficiency(OVector<int> const & code) const
```

Definition: Returns the efficiency (degree of success) with which this chemical could turn on or off an operon with the given switch code. (Delegated to rep)

```
double AvgNBondsToPromoter(OVector<int> const & code) const
```

Definition: Returns the average number of bonds that this chemical would make when binding to the given promoter region. (Delegated to rep)

```
double ProbNoBondToPromoter(OVector<int> const & promoter) const
```

Definition: Returns the probability that this chemical will not immediately bind to the given promoter string. (Delegated to rep)

```
double BindingConstantFor(Chemical const & chem) const
```

Definition: Returns the binding constant for this chemical acting as a carrier for the facilitated diffusion of 'chem'. (Delegated to rep)

```
double AvgNBondsToPermeant(Chemical const & chem) const
```

Definition: Returns the average number of bonds that this chemical would make when binding to the given permeant molecule. (Delegated to rep)

```
int operator<(Chemical const & rhs) const
```

```
int operator<=(Chemical const & rhs) const
```

```
int operator==(Chemical const & rhs) const
```

```
int operator!=(Chemical const & rhs) const
```

```
int operator>=(Chemical const & rhs) const
```

```
int operator>(Chemical const & rhs) const
```

Definition: Comparison operators. Only the shapes are compared, not the amounts.

```
double Match(int const * templ, int length) const
```

Definition: Returns an amount representing the strength of the match between this chemical and the template. This will be a number in [0, 1]. (Delegates to rep)

```
double Match(Chemical const & s1, Chemical const & s2) const
```

Definition: Returns the total catalytic efficiency that this chemical would have when trying to join s1 and s2. (Delegated to rep)

```
AbstractChemicalRep* constructRep(ChemType ct, ChemDigits const & digits, int length, double amount, long maxCount)
```

Definition: Constructs a new rep for the given chemical type. ie, constructs an OrdinaryChemicalRep or an EnzymeChemicalRep, ...

AbstractChemicalRep

This abstract base class represents what is required in all chemicals.

```
class AbstractChemicalRep
{
enum RepType { rt_abstract, rt_chemical, rt_enzyme, rt_partlyExp,
rt_carrier, rt_bound, rt_geneSwitch, rt_water, rt_wall };

    double m_amount;        // concentration

    RepType m_type;        // the type of this rep

    long m_count;          // user count
    long m_maxCount;       // max number of users allowed for this rep

    ChemDigits m_digits;   // shape of chemical
    int m_length;          // number of monomers in shape
};
```

```
virtual void Amount(double newAmt)
```

Definition: Changes the concentration of the rep. Sets `m_amount` to `newAmt`.

```
virtual void MaxCount(long maxCount)
```

Definition: Change the maximum user count.

```
virtual RepType Type() const
```

Definition: Selector returns the rep type: `m_type`.

```
virtual string Name() const
```

Definition: Selector returns the name of the chemical.

```
virtual double Amount() const
```

Definition: Selector returns the concentration of the chemical.

```
virtual ChemDigits const & Digits() const
```

Definition: Selector returns the shape of the chemical: `m_digits`.

```
virtual int Length() const
```

Definition: Selector returns the length of the chemical's shape: `m_length`.

```
virtual bool CanCatalyseReactions() const
```

Definition: At this point, no reps can catalyse reactions. Children will override this function when needed. Returns false.

```
virtual bool CanTakePartInReactions() const
```

Definition: At this stage, returns false. Children will override where necessary.

```
virtual bool AffectsCellWall() const
```

Definition: Returns true if the chemical changes the number of cell wall molecules.

```
virtual bool BuildsCellWall() const
```

Definition: Returns true if the chemical can increase the number of cell wall molecules.

```
virtual bool BreaksCellWall() const
```

Definition: Returns true if the chemical can decrease the number of cell wall molecules. The algorithm is basically the same as for `BuildsCellWall` except that we match against a cell wall breaking molecule.

```
virtual double TranscriptionRate() const
```

Definition: Returns the rate this chemical will allow transcription. The algorithm is the same as for `BuildsCellWall` except that we are matching against a Platonic spider molecule. Also, if the matching is

greater than a threshold we return the amount of matching times the maximum transcription rate, otherwise we return 0.

```
virtual double SwitchEfficiency(OVector<int> const & promoter) const
```

Definition: Returns the degree to which this chemical can switch the given promoter region.

```
double AvgNBondsToPromoter(OVector<int> const & gene) const
```

Definition: Returns the average number of bonds that this chemical would make when binding to the given promoter region. This algorithm is the same as SwitchEfficiency above except that we keep a total of the matchScores instead of the probabilities. We then return this total divided by the number of overlapping positions (cLen + pLen - 1).

```
double ProbNoBondToPromoter(OVector<int> const & gene) const
```

Definition: Returns the probability that this switch chemical will not bind immediately to the given promoter. That is, the number of positions with no match divided by the total number of positions. The algorithm is the same as for AvgNBondsToPromoter except that we count the number of positions that have a matchScore of 0. This is divided, as above, by the number of positions.

```
virtual double BindingConstantFor(AbstractChemicalRep const & acr) const
```

Definition: Returns the binding constant for using this chemical as a carrier for the facilitated diffusion of another rep. At this stage we'll return 0.

```
virtual double AvgNBondsToPermeant(AbstractChemicalRep const & acr) const
```

Definition: Returns the average number of bonds that this chemical would make when binding to a given permeant abstract chemical rep. At this stage we'll return 0 and allow children to override.

```
AbstractChemicalRep(ChemDigits const & digits, int length, double amount, RepType, long maxCount)
```

Definition: Constructor fills in the members that the AbstractChemicalRep knows about, but since it's a protected method, no AbstractChemicalRep objects can be created.

```
AbstractChemicalRep(AbstractChemicalRep const &)
```

Definition: Copy constructor.

```
virtual AbstractChemicalRep* newClone(long maxCount = -1) const = 0
```

Definition: A polymorphic copy constructor. If maxCount is -1 the maxCount for the copy is the same as the old maxCount. maxCount of any other value will be used directly.

```
virtual void Hello()
```

Definition: Registers a user with the rep. Increment m_count.

```
virtual void Goodbye()
```

Definition: Unregisters a user with the rep. Decrement m_count.

```
virtual bool CanRegister() const
```

Definition: Returns true if the rep can take another user. Returns m_maxCount == 0 or (m_maxCount > 0 and m_count < m_maxCount)

```
virtual bool ShouldDelete() const
```

Definition: Returns true if the envelope should delete the rep. Returns m_count == 0.

```
virtual double Match(int const * templ, int length) const
```

Definition: Returns the amount representing the strength of the match between the chemical and the template string. This will be a number in [0, 1]. The algorithm is similar to SwitchEfficiency, above except that we keep account of what the highest number of matches in each position. Finally we return this highest number of matches divided by length of the template.

```
virtual double Match(AbstractChemicalRep const & s1, AbstractChemicalRep const & s2) const
```

Definition: Returns the catalytic efficiency that this chemical would have when trying to join the two given molecules. Since only enzymes can catalyse reactions, we'll return 0 here and override for enzymes.

```
int compare(AbstractChemicalRep const & rhs) const
```

Definition: Compares this rep with another rep and returns -1 if this one is less, 0 if they're equal, and +1 if this one is greater than the other.

OrdinaryChemicalRep

This class is the rep for an ordinary chemical. That is, one that can take part in reactions as a substrate or product, but not as a catalyst. It is publicly derived from AbstractChemicalRep. It adds the extra functionality to AbstractChemicalRep that is needed to model the behaviour of an ordinary chemical.

```
class OrdinaryChemicalRep : public AbstractChemicalRep
{
    // No member variables
};
```

```
virtual bool CanTakePartInReactions() const
```

Definition: Returns true if this rep can act in reactions. Returns true.

```
virtual string Name() const
```

Definition: Returns the name of this rep. Return o followed by the Name the AbstractChemicalRep part returned.

```
OrdinaryChemicalRep(ChemDigits const & digits, int length, double
amount = 0.0, long maxCount = 0)
```

Definition: Constructor.

EnzymeRep

This class is a rep for an enzyme chemical. It is publicly derived from AbstractChemicalRep. It provides the specialisations to AbstractChemicalRep that an enzyme rep requires.

```
class EnzymeRep : public AbstractChemicalRep
{
    // No member variables
};
```

```
virtual bool CanCatalyseReactions() const
```

Definition: Override to return true.

```
virtual string Name() const
```

Definition: Returns the name of the rep. Return e followed by the Name returned by AbstractChemicalRep.

```
EnzymeRep(ChemDigits const & digits, int length, double amount = 0.0,
long maxCount = 0)
```

Definition: Constructor.

```
virtual double Match(AbstractChemicalRep const & s1,
AbstractChemicalRep const & s2) const
```

Definition: Returns the catalytic efficiency that this enzyme would have when joining the two given chemicals.

```
int NumMatches(int pos, OrdinaryChemicalRep const & s1,
OrdinaryChemicalRep const & s2, int& aOverlap, int& bOverlap, int&
aScore, int& bScore) const
```

Definition: Returns the number of matches between this enzyme and two substrates s1 and s2 at position pos. The overlaps on the a and b sides are returned as well.

PartlyExpRep

This class is a rep for a partly expressed protein chemical. It is publicly derived from AbstractChemicalRep. It provides the specialisations to AbstractChemicalRep that a partially expressed protein requires.

```
class PartlyExpRep : public AbstractChemicalRep
{
    // No member variables
};
```

```
virtual string Name() const
```

Definition: Returns the name of the chemical. Return p followed by string returned by the AbstractChemicalRep.

```
virtual bool AffectsCellWall() const
```

Definition: Returns false.

```
virtual bool BuildsCellWall() const
```

Definition: Returns false.

```
virtual bool BreaksCellWall() const
```

Definition: Returns false.

```
PartlyExpRep(ChemDigits const & digits, int length, double amount =
0.0, long maxCount = 0)
```

Definition: Constructor.

CarrierRep

This class is a rep for a carrier chemical. It is derived publicly from AbstractChemicalRep. It provides the specialisations to AbstractChemicalRep that a carrier chemical requires.

```
class CarrierRep : public AbstractChemicalRep
{
    // No member variables
};
```

```
virtual bool AffectsCellWall() const
```

Definition: Returns false.

```
virtual bool BuildsCellWall() const
```

Definition: Returns false.

```
virtual bool BreaksCellWall() const
```

Definition: Returns false.

```
virtual string Name() const
```

Definition: Returns the name of the carrier rep. Return c followed by the string the AbstractChemicalRep::Name returned.

```
virtual double BindingConstantFor(AbstractChemicalRep const & acr)
const
```

Definition: Returns the binding constant for the situation of this chemical acting as a carrier for facilitated diffusion of chem. Just returns Match(acr).

```
virtual double AvgNBondsToPermeant(AbstractChemicalRep const & acr)
const
```


Definition: Returns the number of bonds that this chemical would make when binding to the given permeant.

```
CarrierRep(ChemDigits const & digits, int length, double amount,
RepType, long maxCount)
```

Definition: Constructor.

```
double Match(AbstractChemicalRep const &) const
```

Definition: This method returns the degree of matching between the carrier and another ordinary chemical. This is to calculate the binding constant between the two molecules. The algorithm is the same as AvgNBondsToPermeant above except that instead of summing the match score, we sum a specificity coefficient divided by the probability of getting that match score or higher at random. This is the same as the SwitchEfficiency routine in AbstractChemicalRep.

BoundCarrierRep

This class represents the rep for a bound carrier / permeant complex. It is publicly derived from CarrierRep. It provides the specialisations necessary to CarrierRep to handle the needs of a bound carrier permeant complex. This really means that we just need to override Name.

```
class BoundCarrierRep : public CarrierRep
{
    // No member variables
};
```

```
virtual string Name() const
```

Definition: Returns the name of the rep. Return b followed by the Name returned by AbstractChemicalRep.

```
BoundCarrierRep(ChemDigits const & digits, int length, double amount =
0.0, long maxCount = 0)
```

Definition: Constructor.

GeneSwitchRep

This class represents the rep for an operon switch variable. It is publicly derived from AbstractChemicalRep. It provides the specialisations necessary to AbstractChemicalRep to model an operon switch.

```
class GeneSwitchRep : public AbstractChemicalRep
{
    // No member variables
};
```

```
virtual string Name() const
```

Definition: Returns the name of the rep. Return g followed by the name that AbstractChemicalRep returned.

```
virtual bool AffectsCellWall() const
```

Definition: Return false.

```
virtual bool BuildsCellWall() const
```

Definition: Return false.

```
virtual bool BreaksCellWall() const
```

Definition: Returns false.

```
GeneSwitchRep(ChemDigits const & digits, int length, double amount =
0.0, long maxCount = 0)
```

Definition: Constructor.

WaterRep

This class represents a rep for the number of water molecules. It is publicly derived from AbstractChemicalRep. It provides the specialisations to AbstractChemicalRep necessary to model the number of water molecules.

```
class WaterRep : public AbstractChemicalRep
{
    // No member variables
};
```

```
virtual string Name() const
```

Definition: Returns the name of the rep. Return w followed by the Name returned by AbstractChemicalRep.

```
virtual bool AffectsCellWall() const
```

Definition: Returns false.

```
virtual bool BuildsCellWall() const
```

Definition: Return false.

```
virtual bool BreaksCellWall() const
```

Definition: Return false.

```
WaterRep(ChemDigits const & digits, int length, double amount = 0.0,
long maxCount = 0)
```

Definition: Constructor.

WallRep

This class represents a rep for the number of cell membrane molecules. It is publicly derived from AbstractChemicalRep. It provides the specialisations to AbstractChemicalRep necessary to model the number of cell wall molecules.

```
class WallRep : public AbstractChemicalRep
{
    // No member variables
};
```

```
virtual string Name() const
```

Definition: Returns the name of the rep. Return m followed by the Name returned by AbstractChemicalRep.

```
virtual bool AffectsCellWall() const
```

Definition: Returns false.

```
virtual bool BuildsCellWall() const
```

Definition: Return false.

```
virtual bool BreaksCellWall() const
```

Definition: Return false.

```
WallRep(ChemDigits const & digits, int length, double amount = 0.0,
long maxCount = 0)
```

Definition: Constructor.

Chemistry

This class represents all the metabolic reactions allowable in the simulation. It keeps track of all the possible chemical reactions that enzymes are able to catalyse and generates new reactions that become possible when more chemicals become available. In this latter task, it is important not to generate the same reaction more than once. Only one Chemistry class exists at any one time in the program.

```
class Chemistry
{
    static Chemistry x;           // the one chemistry

    // m_chem:           chemicals that have been used to build
reactions
    // m_pendChem:      chemicals used in the current reaction graph,
    //                  but haven't been used to build other
    //                  reactions.
    // KnownChemicals is the union of these lists.

    OOrder<Chemical>      m_knownChem;

    OOrder<Chemical>      m_chem;
    OOrder<Chemical>      m_pendChem;
    OVector<Reaction>     m_reactions;
    OVector<Reaction>     m_latestReactions;
};
```

```
void Reset()
```

Definition: Restarts the chemistry with no reactions or chemicals.

```
OOrder<Chemical> const & KnownChemicals() const
```

Definition: Selector returns the list of known chemicals.

```
void AddChemicals(OOrder<Chemical> const & newChemicals)
```

Definition: Adds more chemicals to the chemistry and incorporates them into the reaction graph without over counting.

```
OVector<Reaction> const & Reactions() const
```

Definition: Selector returns a list of the currently valid metabolic reactions.

```
OVector<Reaction> const & LatestReactions() const
```

Definition: Selector returns a list of the reactions added since the last call to AddChemicals.

```
Chemistry()
```

Definition: Constructor.

```
void AddCondensation( participants );
```

Name: AddCondensation

Definition: Adds new condensation reactions, taking each participant in the reaction from the respective list passed in.

```
void AddCleavage( participants );
```

Definition: Adds new cleavage reactions to the chemistry, taking each chemical from the lists passed in.

Differential equation classes

CellDiffEqu

This class represents a differential equation used in the cell. Like the Chemical class, we're using an envelope / letter class idiom to make the classes more efficient. Again, the rep (envelope) pointer is polymorphic. We don't use pointer counting here, though. The responsibilities of this class are to hold the polymorphic DE rep and to delegate to it.

```
class CellDiffEqu
{
    enum VarType { vt_ord, vt_ord_bar, vt_enz, vt_enz_bar, vt_pex,
vt_car, vt_bnd, vt_water, vt_wall, vt_switch };

    DERep*    m_rep;
};
```

```
CellDiffEqu(VarType = vt_ord, Chemical const & var = kNoChem, TwoDPos
const & xidx = kNoPos, TwoDPos const & xbaridx = kNoPos, World* pWorld
= 0, ParsedOperon::SwitchStyle ss = ParsedOperon::ss_alwaysOn,
OVector<int> const & gene = kEmptyVector)
```

Definition: Constructor. Switches on the variable type passed in to construct the appropriate DERep and set m_rep to point to it. VarType is the type of variable being modelled, var is the actual variable, xidx is the index of the variables in the values array for the x variable (ie, in m_vars), xbaridx is the index of the invented variable (x bar i) in the values array, world pointer points the environment, ss is the type of switching used for this switch DE (ignored for other DEs), and gene is the switch / promoter region for the switch DE (also ignored for other DEs).

```
CellDiffEqu(CellDiffEqu const &)
```

Definition: Copy constructor. Initialise m_rep with a newClone of the m_rep in the other CellDiffEqu.

```
CellDiffEqu const & operator=(CellDiffEqu const &)
```

Definition: Assignment operator. Delete the current m_rep then set m_rep to the newClone of the other CellDiffEqu.

```
~CellDiffEqu()
```

Definition: Destructor. Delete the m_rep.

```
Chemical const & Var() const
```

Definition: Selector returns the variable this differential equation represents. (delegate to the rep)

```
TwoDPos const & X() const
```

Definition: Selector returns the index of the x variable in the variables array. (delegate to rep)

```
double operator()(TwoDBArr<Chemical> const & vals, World const &,
TwoDBArr<CellDiffEqu> const & des) const
```

Definition: Function operator. Calculates and returns the value of the differential equation given a set of values for the variables. (delegates to rep)

```
double InitialValue(TwoDBArr<Chemical> const & vals,
TwoDBArr<CellDiffEqu> const & des) const
```

Definition: Returns an initial value for the variable governed by the differential equation. (delegated to rep)

```
void ConsiderChemical(Chemical const & chem, TwoDPos const & itsPosn,
double switchEfficiency)
```

Definition: Changes the differential equation given exposure to a new chemical. (delegated to rep)

```
void ConsiderReaction(IdxReaction<TwoDPos> const &)
```

Definition: Given a metabolic reaction, modify the terms in the differential equation appropriately. (delegated to rep)

```
void ConsiderReaction(TransReaction const &)
```

Definition: Given a transcription reaction, modify the terms in the differential equation appropriately. (delegated to rep)

```
void ConsiderReaction(EnzymeDegradationReaction const &, long
countOfThisMonomerInTheEnzyme = 0)
```

Definition: Given an enzyme degradation reaction, modify the terms in the differential equation appropriately. (delegated to rep)

```
void ConsiderReaction(DiffusionReaction const &)
```

Definition: Given a facilitated diffusion reaction, modify the terms in the differential equation appropriately. (delegated to rep)

DERep

This class is an abstract base class for a rep for differential equations. It provides the interface for all methods required in reps for differential equations.

```
class DERep
{
    Chemical m_var;           // the variable being modelled
    TwoDPos m_xidx;          // index of x in the values array

    OVector<CellReactTerm> m_terms; // a list of the reaction terms of
                                   // the differential equation
};
```

```
DERep(Chemical const & var = kNoChem, TwoDPos const & xidx = kNoPos)
```

Definition: Constructor. var is the variable and xidx is its index in the values array.

```
virtual DERep* newClone() = 0
```

Definition: Virtual copy constructor.

```
virtual Chemical const & Var() const
```

Definition: Selector returns the variable for the differential equation.

```
virtual TwoDPos const & X() const
```

Definition: Selector returns the index in the variables array of the variable for the differential equation.

```
virtual double operator()(TwoDBArr<Chemical> const & vals, World const
&, TwoDBArr<CellDiffEqu> const & des) const
```

Definition: Returns the value of the differential equation for a given set of values.

```
virtual double InitialValue(TwoDBArr<Chemical> const & vals,
TwoDBArr<CellDiffEqu> const & des) const
```

Definition: Returns an initial value for the variable. At this stage, just return 0.

```
virtual void ConsiderChemical(Chemical const &, TwoDPos const &,
double)
```

Definition: By default ignore the chemical.

```
virtual void ConsiderReaction(IdxReaction<TwoDPos> const &)
```

Definition: Adjusts the differential equation given a metabolic equation.

```
virtual void ConsiderReaction(TransReaction const &)
```

Definition: Adjusts the differential equation given a transcription reaction.

```
virtual void ConsiderReaction(EnzymeDegradationReaction const &, long
count = 0)
```

Definition: Looks at an enzyme degradation reaction and adds terms to the differential equation if necessary.

```
virtual void ConsiderReaction(DiffusionReaction const &)
```

Definition: Looks at a facilitated diffusion reaction and adds terms to the differential equation if necessary - but at this level just ignore diffusion reactions.

```
virtual OVector<CellReactTerm> const & Terms() const
```

Definition: Returns a read only list of the terms in the differential equation.

```
virtual OVector<CellReactTerm>& Terms()
```

Definition: Overloaded version of the previous function that returns a writeable list of terms for children.

GeneUtil

This is a very simple class representing the utilisation of the operon by one switch chemical. That is, the amount of time that a switch chemical binds to the promoter region of the operon. Note that it doesn't take into account the other switch chemicals that are also present. That's taken care of in GeneSwitchDERep. It keeps track of the operon utilisation by one species of switch chemical. This involves calculation of the utilisation for a particular concentration of the chemical and calculation of the derivative of the operon utilisation at a given concentration.

```
class GeneUtil
{
    TwoDPos m_chemi; // the index of the switch chemical in the arrays
    double m_coeff; // a coefficient used to calculate the value and
                  // deriv
};
```

```
GeneUtil()
```

Definition: Default constructor.

```
GeneUtil(TwoDPos const & i, Chemical const & c, OVector<int> const &
gene, double efficiency)
```

Definition: Constructor. *i* is the index of the switch chemical, *c* is the switch chemical, *gene* is the promoter region, and *efficiency* is how well the switching chemical works.

```
double Value(TwoDBArr<Chemical> const & vars) const
```

Definition: Returns the operon utilisation given the concentration of the switch chemical.

```
double Deriv(TwoDBArr<CellDiffEqu> const & des, World const & w,
TwoDBArr<Chemical> const & vars) const
```

Definition: Returns the derivative of the operon utilisation with respect to time, given the derivative of the concentration of switch chemical with respect to time.

GeneSwitchDERep

This class is the rep for the differential equation governing an operon switch variable. It is publicly derived from DERep. It provides the specialisations to DERep that are necessary to model an operon switch differential equation.

```
class GeneSwitchDERep : public DERep
{
    ParsedOperon::SwitchStyle m_ss;
    BucketArray<GeneUtil> m_utils;
    OVector<int> m_gene;
};
```

```
GeneSwitchDERep(Chemical const & var, TwoDPos const & xidx,
ParsedOperon::SwitchStyle, OVector<int> const & gene)
```

Definition: Constructor.

```
virtual double operator()(TwoDBArr<Chemical> const & vals, World const
&, TwoDBArr<CellDiffEqu> const & des) const
```

Definition: Calculates and returns the value of the DE given a set of values.

```
virtual double InitialValue(TwoDBArr<Chemical> const & vals,
TwoDBArr<CellDiffEqu> const & des) const
```

Definition: Returns a suggested initial value for the variable associated with this differential equation.

```
virtual void ConsiderChemical(Chemical const &, TwoDPos const &,
double)
```

Definition: Given the new chemical, modify the differential equation accordingly.

```
virtual void ConsiderReaction(IdxReaction<TwoDPos> const &)
```

Definition: Given a metabolic reaction, change the differential equation accordingly. Here, this means do nothing: reactions don't have any effect on switch variables.

```
virtual void ConsiderReaction(TransReaction const &)
```

Definition: Given a transcription reaction, change the differential equation accordingly. Here, this means do nothing: reactions don't have any effect on switch variables.

```
double UtilDeriv(TwoDBArr<Chemical> const & vals, World const & w,
TwoDBArr<CellDiffEqu> const & des) const
```

Definition: Calculate the combined utilisation derivative over all of the switch chemicals.

```
double UtilVal(TwoDBArr<Chemical> const & vals, TwoDBArr<CellDiffEqu>
const & des) const
```

Definition: Calculates the combined value over all the switch chemicals.

BagleyDERep

This class is a rep for a Bagley style differential equation. That is, one that takes part in the metabolic reactions, with the bound forms of the variable. At this stage it's an abstract base class. It is publicly derived from DERep. It provides the specialisations to DERep necessary to model the Bagley style reactions.

```
class BagleyDERep : public DERep
{
    // BVarType is the kind of variable the differential equation deals
    // with: an ordinary x variable or the invented variable x bar.
    enum BVarType { x, xbar };

    BVarType m_vtype;        // the type of variable being modelled
    TwoDPos m_xbaridx;      // index of x bar i, in the values array
};
```

```
BagleyDERep(BVarType = x, Chemical const & var = DERep::kNoChem,
TwoDPos const & xidx = DERep::kNoPos, TwoDPos const & xbaridx =
DERep::kNoPos)
```

Definition: Constructor. BVarType is the type of variable being modelled, var is the variable, xidx is the index of the variable in the values array for the x variable and xbaridx is the index of the invented variable (x bar i) in the values array.

```
virtual double operator()(TwoDBArr<Chemical> const & vals, World const
&, TwoDBArr<CellDiffEqu> const & des) const
```

Definition: Calculates and returns the value of the differential equation given a set of values.

```
virtual void ConsiderReaction(IdxReaction<TwoDPos> const &)
```

Definition: Given a metabolic reaction change the differential equation to reflect it.

```
virtual void ConsiderReaction(TransReaction const &)
```

Definition: Change the differential equation to reflect a transcription reaction.

```
virtual BVarType VariableType() const
```

Definition: Selector returns the type of this variable - unbound or bound.

```
virtual TwoDPos const & XBar() const
```

Definition: Selector returns the index of the bound version of the variable.

OrdChemDERep

This class is a rep for the differential equation of an ordinary chemical that is used and produced in reactions and can diffuse through the cell membrane. It is publicly derived from BagleyDERep. It provides the specialisations necessary to BagleyDERep to model an ordinary chemical's differential equation.

```
class OrdChemDERep : public BagleyDERep
{
    World* m_pWorld; // access to the outside of the cell for diffusion
                    // purposes.
};
```

```
OrdChemDERep(BVarType = x, Chemical const & var = DERep::kNoChem,
TwoDPos const & xidx = DERep::kNoPos, TwoDPos const & xbaridx =
DERep::kNoPos, World* pWorld = 0)
```

Definition: Constructor. BVarType is the type of variable being modelled, var is the variable being modelled, xidx is the index of the variable in the values array for the x variable, xbaridx is the index of the invented variable (x bar i) in the values array and World* is access to the outside of the cell (not necessary for bar variables).

```
virtual double operator()(TwoDBArr<Chemical> const & vals, World const
&, TwoDBArr<CellDiffEqu> const & des) const
```

Definition: Calculates the value of the differential equation for a given set of variables.

```
virtual void ConsiderReaction(DiffusionReaction const &)
```

Definition: Change the differential equation to reflect a facilitated diffusion reaction.

EnzymeDERep

This class is a rep for the differential equation for an enzyme variable. Enzymes can't diffuse through the membrane. It is derived from BagleyDERep. It provides the specialisations to BagleyDERep necessary to model a differential equation for an enzyme variable. There really aren't any specialisations to make. EnzymeDERep is just a wrapper for BagleyDERep.

```
class EnzymeDERep : public BagleyDERep
{
    // No member variables
};
```

```
EnzymeDERep(BVarType = x, Chemical const & var = DERep::kNoChem,
TwoDPos const & xidx = DERep::kNoPos, TwoDPos const & xbaridx =
DERep::kNoPos)
```

Definition: Constructor. BVarType is the type of variable being modelled, var is the variable being modelled, xidx is the index of the variable in the values array for the x variable and xbaridx is the index of the invented variable (x bar i) in the values array.

WaterDERep

This class is a rep for a differential equation governing the number of water molecules in the cell. It is derived publicly DERep. It provides specialisations to the DERep class to model a differential equation governing the number of water molecules in a cell.


```
class WaterDERep : public DERep
{
    double m_k;
};
```

```
WaterDERep(Chemical const & var = DERep::kNoChem, TwoDPos const & xidx
= DERep::kNoPos)
```

Definition: Constructor.

```
virtual double InitialValue(TwoDBArr<Chemical> const & vals,
TwoDBArr<CellDiffEqu> const & des) const
```

Definition: Calculates the initial value for the number of water molecules variable given a set of values for the variables. The initial value is $\frac{2}{3} * m_k * (Nm ** 1.5)$, where Nm is the number of cell wall molecules in the cell (read from vals).

```
virtual double operator() (TwoDBArr<Chemical> const & vals, World const
&, TwoDBArr<CellDiffEqu> const & des) const
```

Definition: Calculates the value of the differential equation given a set of values.

```
virtual void ConsiderChemical(Chemical const &, TwoDPos const &,
double)
```

Definition: Changes the differential equation given a new chemical. Here, we do nothing.

```
virtual void ConsiderReaction(IdxReaction<TwoDPos> const &)
```

Definition: Changes the DE given a metabolic reaction. Here, there is nothing to do.

```
virtual void ConsiderReaction(TransReaction const &)
```

Definition: Changes the DE given a transcription reaction. Here, there is nothing to do.

WallDERep

This class is a rep for a differential equation governing the variable for the number of cell membrane molecules in a cell. It is publicly derived from DERep. It provides the specialisations to DERep needed for the differential equation for the number of cell membrane molecules.

```
class WallDERep : public DERep
{
    // Positions of chemicals used to modify the number of cell wall
    // molecules
    OVector<TwoDPos> m_builders;
    OVector<TwoDPos> m_breakers;
};
```

```
WallDERep(Chemical const & var = DERep::kNoChem, TwoDPos const & xidx
= DERep::kNoPos)
```

Definition: Constructor.

```
virtual double InitialValue(TwoDBArr<Chemical> const & vals,
TwoDBArr<CellDiffEqu> const & des) const
```

Definition: Returns a suggested initial value for the variable. In this case, just get the value out of vals and return it.

```
virtual double operator() (TwoDBArr<Chemical> const & vals, World const
&, TwoDBArr<CellDiffEqu> const & des) const
```

Definition: Calculates and returns the value of the differential equation given a particular set of variable values.

```
virtual void ConsiderChemical(Chemical const &, TwoDPos const &,
double)
```

Definition: Changes the differential equation given a newly discovered chemical species.

```
virtual void ConsiderReaction(IdxReaction<TwoDPos> const &)
```

Definition: Ignored as reactions don't directly affect the number of cell wall molecules.

```
virtual void ConsiderReaction(TransReaction const &)
```

Definition: Ignored as transcription reactions don't directly affect the number of cell wall molecules.

PartlyExpDERep

This class is a rep for a differential equation governing partially expressed protein variables. It is publicly derived from DERep. It provides the specialisations necessary to DERep to model the differential equation for partially expressed proteins.

```
class PartlyExpDERep : public DERep
{
    // No member variables
};
```

```
PartlyExpDERep(Chemical const & var = DERep::kNoChem, TwoDPos const &
xidx = DERep::kNoPos)
```

Definition: Constructor.

```
virtual void ConsiderReaction(IdxReaction<TwoDPos> const &)
```

Definition: Changes the differential equation based on a metabolic reaction. In this case, ignore the reaction. ie, don't do what DERep would have us do.

CarrierDERep

This class is a rep for a differential equation for carrier species. It is publicly derived from DERep. It provides the specialisations to DERep necessary to model carrier differential equations.

```
class CarrierDERep : public DERep
{
    // No member variables
};
```

```
CarrierDERep(Chemical const & var = DERep::kNoChem, TwoDPos const &
xidx = DERep::kNoPos)
```

Definition: Constructor.

```
virtual void ConsiderReaction(IdxReaction<TwoDPos> const &)
```

Definition: Changes the differential reaction given a metabolic reaction. In this class, that means, do nothing. Carriers aren't directly affected by metabolic reactions.

```
virtual void ConsiderReaction(DiffusionReaction const &)
```

Definition: Changes the differential equation given a facilitated diffusion reaction.

BoundCarrierDERep

This class is a rep for a differential equation governing a bound carrier / permeant species. It is derived publicly from CarrierDERep. It provides the specialisations to CarrierDERep needed to model a differential equation for bound carrier / permeant molecules.

```
class BoundCarrierDERep : public CarrierDERep
{
    // No member variables
};
```

```
BoundCarrierDERep(Chemical const & var = DERep::kNoChem, TwoDPos const
& xidx = DERep::kNoPos)
```

Definition: Constructor.

```
virtual void ConsiderReaction(DiffusionReaction const &)
```

Definition: Changes the differential equation given a particular facilitated diffusion reaction.

CellReactTerm

This class represents a term in one of the differential equations for the chemicals. It defines and calculates the value of a term in a differential equation. From an historical viewpoint, the design of this class is something of a disaster. I have identified five different kinds of term required for the chemical differential equations. Rather than use polymorphism, I've crammed all five kinds of term into this one class. This is an historical artifact. The terms are for metabolic reactions, transcription reactions, enzyme degradation reactions, and two kinds of terms for facilitated diffusion terms.

```
class CellReactTerm
{
    enum PosNeg { positive, negative };

    enum ReactionType { rt_ordinary, rt_transcription, rt_enz_decay,
rt_diffusion, rt_diffusion2 };

    ReactionType m_rt;    // the kind of term we're dealing with
    double m_k;          // the reaction rate
    double m_v;          // the enzyme efficacy
    TwoDPos m_eidx;      // idx of the enzyme in the values array
    TwoDPos m_aidx;      // idx of the substrate A in the values array
    TwoDPos m_bidx;      // idx of the substrate B in the values array
    Chemical m_chem;
};
```

```
CellReactTerm(PosNeg, double k, double v, TwoDPos const & eidx,
TwoDPos const & aidx, TwoDPos const & bidx)
```

Definition: Constructor for an ordinary reaction term (metabolic reaction). PosNeg says whether the term is positive or negative, k is the reaction rate, v is the enzyme efficacy, eidx is the index of the enzyme concentration, aidx is the index of the substrate A concentration and bidx is the index of the substrate B concentration.

```
CellReactTerm(PosNeg, double k, TwoDPos const & sidx, TwoDPos const &
tidx, TwoDPos const & bidx)
```

Definition: Constructor for a transcription reaction term. PosNeg says whether the term is positive or negative, k is the transcription rate, sidx is the index of the switch variable, tidx is the index of the spider (transcription agent) concentration and bidx is the index of the base concentration.

```
CellReactTerm(PosNeg, double k, TwoDPos const & eidx)
```

Definition: Constructor for enzyme degradation terms. PosNeg says whether the term is positive or negative, k is the decay rate * number of monomers of this type and eidx is the index of the switch variable

```
CellReactTerm(PosNeg, double k, Chemical const & pe, TwoDPos const &
pi_idx, TwoDPos const & bidx)
```

Definition: Constructor for the first kind of facilitated diffusion term. PosNeg says whether the term is positive or negative, k is the diffusion constant, pe is the name of the permeant chemical, pi_idx is the index of the internal permeant chemical species and bidx is the index of the bound chemical species in the cell.

```
CellReactTerm(PosNeg, double k, Chemical const & pe, TwoDPos const &
cidx)
```

Definition: Constructor for the second kind of facilitated diffusion term. PosNeg says whether the term is positive or negative, k is the diffusion constant, pe is the name of the permeant chemical and cidx is the index of the internal carrier chemical species.

```
double operator() (TwoDBArr<Chemical> const & vals, World const & w)
const
```

Definition: Function operator. Calculates the value of the term.

```
CellReactTerm const & Reset(PosNeg, double k, double v, TwoDPos const
& eidx, TwoDPos const & aidx, TwoDPos const & bidx)
```

Definition: Reconstructor for metabolic reactions. Acts similarly to the matching constructor, but doesn't allocate memory.

```
CellReactTerm const & Reset(PosNeg, double k, TwoDPos const & sidx,
TwoDPos const & tidx, TwoDPos const & bidx)
```

Definition: Reconstructor for transcription reactions. Acts similarly to the matching constructor, but doesn't allocate memory.

```
CellReactTerm const & Reset(PosNeg, double k, TwoDPos const & eidx)
```

Definition: Reconstructor for enzyme degradation reactions. Acts similarly to the matching constructor, but doesn't allocate memory.

```
CellReactTerm const & Reset(PosNeg, double k, Chemical const & pe,
TwoDPos const & pi_idx, TwoDPos const & bidx)
```

Definition: Reconstructor for facilitated diffusion (1) reactions. Acts similarly to the matching constructor, but doesn't allocate memory.

```
CellReactTerm const & Reset(PosNeg, double k, Chemical const & pe,
TwoDPos const & cidx)
```

Definition: Reconstructor for facilitated diffusion (2) reactions. Acts similarly to the matching constructor, but doesn't allocate memory.

Reaction classes

Reaction

This class models a metabolic reaction: $s_1 + s_2 \rightleftharpoons p$ catalysed by cat.

```
class Reaction
{
    Chemical m_s1;
    Chemical m_s2;
    Chemical m_p;
    Chemical m_cat;

    double m_kf;    // forward rate
    double m_kr;    // reverse rate
    double m_v;     // catalytic efficiency
};
```

```
Reaction(Chemical const & s1, Chemical const & s2, Chemical const &
cat, double v)
```

Definition: Constructor. s1 and s2 are the substrate chemicals, cat is the catalyst and v is the catalytic efficiency.

```
virtual Chemical const & Sub1() const
virtual Chemical const & Sub2() const
virtual Chemical const & Product() const
virtual Chemical const & Catalyst() const
```

Definition: Selectors. Return the chemicals involved.

```
virtual double ForwardRate() const
virtual double ReverseRate() const
virtual double UnbindingRate() const
virtual double EnzymeEfficiency() const
```

Definition: Selectors.

```
virtual int Involves(Chemical const & c) const
```

Definition: Returns 1 if a particular chemical is involved in the reaction.

IdxReaction

A parameterised class (on an Index class :often TwoDPos). It represents a reaction, but also knows the indices (in the cell's variable and DE arrays) of each of the chemicals involved. It is derived publicly from Reaction. Its main responsibility is to deal with indices of the chemical species in metabolic reactions. A constraint with this class is that the Index class used as a parameter must have a default constructor, copy constructor, assignment operator, equivalence operator, and output to an output stream defined.

```
template <class Index>
class IdxReaction : public Reaction
{
    Index m_s1i;    // substrate 1 index
    Index m_s2i;    // substrate 2 index
    Index m_pi;     // product index
    Index m_wi;     // water index
    Index m_cati;   // catalyst index
};
```

```
IdxReaction(Chemical const & s1, Index const & s1i, Chemical const &
s2, Index const & s2i, Index const & prodi, Index const & wateri,
Chemical const & cat, Index const & cati, double v)
```

Definition: Constructor.

```
IdxReaction(Reaction const & r, Index const & s1i, Index const & s2i,
Index const & prodi, Index const & wateri, Index const & cati)
```

Definition: Constructor (from an existing reaction)

```
virtual Index const & Sub1Idx() const
virtual Index const & Sub2Idx() const
virtual Index const & ProductIdx() const
virtual Index const & WaterIdx() const
virtual Index const & CatalystIdx() const
```

Definition: Selectors for each of the indices.

TransReaction

This class represents a transcription reaction. The following 5 reactions are possible:

- (1) Spider + Base \rightarrow PExp kt, switch
- (2) Spider + Base \rightarrow Protein kt, switch
- (3) PExp + Base \rightarrow PExp kt
- (4) PExp + Base \rightarrow PExp + Protein kt
- (5) PExp + Base \rightarrow Spider + Protein kt

The protein resulting will be either an enzyme or a carrier. Spiders and PExps are transcription agents \rightarrow tagent

```
class TransReaction
```

```

{
  Chemical m_tagent1;
  Chemical m_base;
  Chemical m_tagent2;
  Chemical m_enzyme;

  TwoDPos m_tagent1i;
  TwoDPos m_basei;
  TwoDPos m_tagent2i;
  TwoDPos m_enzymei;

  double    m_kt;
  TwoDPos   m_switchi;
};

```

TransReaction(Chemical const & tagent1, TwoDPos const & tagent1i, Chemical const & base, TwoDPos const & basei, Chemical const & tagent2, TwoDPos const & tagent2i, Chemical const & enzyme = kNoChem, TwoDPos const & enzymei = kNoPos, double kt = 0, TwoDPos const & switchi = kNoPos)

Definition: Creates a transcription reaction. Any chemicals that don't exist in the reaction should be given as kNoChem and their indices as kNoPos.

```

Chemical const & TAgent1() const
Chemical const & Base() const
Chemical const & TAgent2() const
Chemical const & Enzyme() const

```

Definition: Selectors for each of the chemical species involved in the reaction.

```

TwoDPos const & TAgent1Idx() const
TwoDPos const & BaseIdx() const
TwoDPos const & TAgent2Idx() const
TwoDPos const & EnzymeIdx() const

```

Definition: Selectors for the indices of each of the species involved in the reaction.

```

double TranscriptionRate() const

```

Definition: Selector returns the rate constant to be used for the reaction.

```

TwoDPos const & SwitchIdx() const

```

Definition: Selector returns the index of the switching variable for the reaction.

```

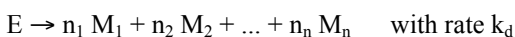
int Involves(Chemical const & c) const

```

Definition: Returns 1 if chemical c is involved in the reaction, 0 otherwise.

EnzymeDegradationReaction

This class represents a reaction modelling the degradation of a protein into its constituent monomers. This class is valid for carriers as well as enzymes. It models the following reaction:



An enzyme breaks down into $n_i M_i$ molecules, where n_i is the number of M_i monomers in the enzyme with reaction rate constant k_d .

eg. e0121 \rightarrow o0 + 2 o1 + o2

```

class EnzymeDegradationReaction
{
  Chemical m_enzyme;
  TwoDPos m_enzymei;
  double m_kd;
};

```

EnzymeDegradationReaction(Chemical const & enz, TwoDPos const & enzi)
 Definition: Constructor. Reads the decay rate constant from the cell.par parameter file.

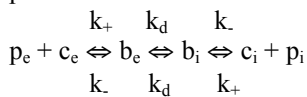
Chemical const & Enzyme() const
 Definition: Selector. Returns the protein species being degraded.

TwoDPos const & EnzymeIdx() const
 Definition: Selector. Returns the index of the protein being degraded.

double DecayRate() const
 Definition: Selector. Returns the rate constant for the degradation reaction.

DiffusionReaction

This class represents a facilitated diffusion reaction - ie, a set of reactions governing the transport of a permeant across the cell membrane using a carrier molecule. It models the following family of reactions:



p_e is the external concentration of permeant
 c_e is the concentration of carrier on the outer side of the membrane
 b_e is the concentration of bound carrier on the outer side of the membrane
 b_i is the concentration of bound carrier on the inner side of the membrane
 c_i is the concentration of carrier inside the cell
 p_i is the concentration of permeant inside the cell
 k_+ is the rate constant determining carrier / permeant binding
 k_- is the rate constant determining permeant release
 k_d is the rate constant determining bound carrier diffusion through the membrane. It's not a real rate constant in the manner of chemical kinetic rate constants.

```
class DiffusionReaction
{
  Chemical m_permeant_i;
  TwoDPos m_permeant_ii;
  Chemical m_carrier_i;
  TwoDPos m_carrier_ii;
  Chemical m_bound_i;
  TwoDPos m_bound_ii;
  double m_k_plus;
  double m_k_minus;
};
```

```
DiffusionReaction(Chemical const & permeant_i, TwoDPos const &
permeant_ii, Chemical const & carrier_i, TwoDPos const & carrier_ii,
Chemical const & bound_i, TwoDPos const & bound_ii, double k_plus,
double k_minus)
```

Definition: Constructor. permeant_i is the internal permeant chemical, permeant_ii is the index of the internal permeant chemical, carrier_i is the internal carrier molecule, carrier_ii is the index of the internal carrier chemical, bound_i is the internal bound carrier / permeant complex, bound_ii is the index of the internal bound carrier / permeant complex, k_plus is the forward binding rate constant and k_minus is the reverse binding rate constant.

Chemical const & Permeant() const
 Definition: Selector returns the permeant chemical.

TwoDPos const & PermeantIIdx() const
 Definition: Selector returns the index of the permeant chemical in the variable and DEs array.

Chemical const & Carrier() const

Definition: Selector returns the carrier species.

```
TwoDPos const & CarrierIIdx() const
```

Definition: Selector returns the index of the carrier chemical in the variable and DEs array.

```
Chemical const & Bound() const
```

Definition: Selector returns the bound carrier / permeant species.

```
TwoDPos const & BoundIIdx() const
```

Definition: Selector returns the index of the bound carrier / permeant chemical in the variable and DEs array.

```
double KPlusRate() const
```

Definition: Selector returns the forward rate constant for binding the permeant and carrier into the bound complex.

```
double KMinusRate() const
```

Definition: Selector returns the reverse rate constant for unbinding the bound complex into a permeant and carrier molecule.

Support classes (others are not given)

BucketArray

This is a parameterised class (parameterised on 'type'). It acts like a normal fixed size C++ array except that it is allocated in smaller blocks instead of one huge memory block. The aim is to avoid memory fragmentation that occurs when trying to allocate very large arrays. Elements going into a BucketArray need a default constructor and this is called N times on the construction of the BucketArray. Its main responsibility is to act as similarly to an ordinary array as possible. A constraint is that classes put into the BucketArray must have a default constructor, copy constructor and assignment operator defined.

```
template <class type>
class BucketArray
{
    type**    m_buckets;           // the array of buckets
    long      m_numElems;         // the number of elements in the array
    long      m_bucketSize;       // number of elements per bucket
    long      m_numBuckets;       // the number of buckets in m_buckets.
};
```

```
BucketArray(long elements, long bucketSize = 0)
```

Definition: Constructor creates an array of 'elements' number of items in buckets of size 'bucketSize'. The default bucket size is 1000.

```
long Length() const
```

Definition: returns the number of elements in the BucketArray.

```
type const & operator[](long idx) const
```

Definition: Square brackets operator used to get read only access to elements of the array.

```
type& operator[](long idx)
```

Definition: Square brackets operator used to get access to elements of the array. Elements may be modified.

TwoDPos

This class represents an index in a TwoDBArr, ie. a two dimensional bucket array.

```
class TwoDPos
```



```
{
    long m_x;          // primary index - ie. row
    long m_y;          // secondary index - ie. col
};
```

```
TwoDPos(long x = -1, long y = -1)
```

Definition: Constructor creates an index for position (x, y).

```
long X() const
```

Definition: Returns the x part of the coordinate.

```
long Y() const
```

Definition: Returns the y part of the coordinate.

TwoDBArr

This parameterised class (templated on 'type') is a two dimensional bucket array, ie. a jagged array. It handles allocation and storage of elements in a jagged array whilst keeping an interface as similar as a C++ array as possible.

```
template <class type>
class TwoDBArr
{
    enum { kBucketSize = 100 };
    BucketArray<BucketArray<type>*> m_rows;
};
```

```
TwoDBArr(long rows)
```

Definition: Constructor creates a two dimensional bucket array containing 'rows' number of rows. The default bucket size for the rows will be 100.

```
long Rows() const
```

Definition: Returns the number of rows in the TwoDBArr.

```
long Cols(long row) const
```

Definition: Returns the number of columns in the 'row'th row.

```
void AllocateRow(long row, long length)
```

Definition: Allocate 'elements' number of elements in the row'th row. This has no effect on rows that have already been allocated. (See ReAllocateRow).

```
void ReAllocateRow(long row, long length)
```

Definition: Reallocate the row'th row to be 'length' elements long. The old elements are copied over. If the row is shortened the remaining elements are destroyed.

```
type const & Item(long idx, long jdx) const
```

Definition: Returns a read only version of the element at row idx and column jdx.

```
type& Item(long idx, long jdx)
```

Definition: Returns a writeable version of the element a row idx and column jdx.

```
type const & Item(TwoDPos const & p) const
```

Definition: Returns a read only version of the element at coordinate p.

```
type& Item(TwoDPos const & p)
```

Definition: Returns a writeable version of the element at coordinate p.

REFERENCES

- Ackley, D. H. and Littman, M. L., 1993, "A Case for Lamarckian Evolution" in "Artificial Life III", edited by Langton, C. G., SFI Studies in the Sciences of Complexity, Proc. Vol. XVII, Addison-Wesley.
- Aho, A. V., Sethi, R. and Ullman, J. D., 1986, "Compilers: Principles, Techniques and Tools", Addison-Wesley.
- Alberts, Bray, et al. 1994, "Molecular Biology of the Cell", 3rd edition, Garland Publishing: New York
- Avery, H. E., 1974, "Basic Reaction Kinetics and Mechanisms", MacMillan Education: London.
- Back, T., 1996, "Evolutionary Algorithms in Theory and Practice: evolution strategies, evolutionary programming, genetic algorithms", Oxford University Press: New York.
- Bagley, R. J. and Farmer, J. D., 1991, "Spontaneous Emergence of a Metabolism" in Artificial Life II, SFI Studies in the Sciences of Complexity, vol. X, edited by Langton, C.G., Farmer, J. D. and Rasmussen, S., Addison-Wesley.
- Bagley, R. J., Farmer, J. D. and Fontana, W., 1991, "Evolution of a Metabolism" in Artificial Life II, SFI Studies in the Sciences of Complexity, vol. X, edited by Langton, C.G., Farmer J. D. and Rasmussen S., Addison-Wesley.
- Banzhaf, W., Dittrich, P., and Rauhe, H., 1996, "Emergent Computation by Catalytic Reactions", Nanotechnology, 7 pp. 1-8.
- Booch, G., 1994, "Object-Oriented Analysis and Design with Applications", 2nd edition, Benjamin/Cummings Publishing: Redwood City, California.
- Brookshear, J. G., 1989, "Theory of Computation: Formal Languages, Automata, and Complexity", Benjamin/Cummings Publishing: Redwood City, California.
- Chaplin, M. F. and Bucke, C., 1990, "Enzyme Technology", Cambridge University Press.
- Coplien, J. O., 1994, "Advanced C++ Programming Styles and Idioms ", Addison-Wesley.
- Dawkins, R., 1986, "The Blind Watchmaker", Penguin.

- Dawkins, R., 1988, "The Evolution of Evolvability" in *Artificial Life*, SFI Studies in the Sciences of Complexity, vol. VI, edited by Langton C.G., Addison-Wesley.
- Dawkins, R., 1989, "The Selfish Gene", New edition, Oxford University Press.
- Farmer, J. D., Kauffman, S. A. and Packard, N. H., 1986, "Autocatalytic Replication of Polymers", *Physica* 22D pp. 50-67.
- Farmer, J. D., Packard, N. H. and Perelson, A. S., 1986, "The Immune System, Adaptation and Machine Learning", *Physica* 22D p187.
- Fleischer, K. and Barr, A. H., 1994, "A Simulation Testbed for the Study of Multicellular Development: The Multiple Mechanisms of Morphogenesis" in "Artificial Life III", edited by Langton, C. G., SFI Studies in the Sciences of Complexity, Proc. Vol. XVII, Addison-Wesley.
- Gardner, E. J., Simmons, M. J. and Snustad, D. P., 1991, "Principles of Genetics", 8th ed., John Wiley & Sons.
- Goldberg, D. E., 1989, "Genetic Algorithms in Search, Optimization and Machine Learning", Addison-Wesley.
- Hertz, J., Krogh, A., Palmer, R. G., 1991, "Introduction to the Theory of Neural Computation", Addison-Wesley.
- Hofbauer, J. and Sigmund, K., 1988, "The Theory of Evolution and Dynamical Systems: Mathematical Aspects of Selection", Cambridge University Press: Cambridge.
- Holcombe, M., 1994, "Chapter 2: From VLSI through Machine Models to Cellular Metabolism" in "Computing with Biological Metaphors", edited by Paton, R., Chapman and Hall: London.
- Holland, J. H., 1992, "Adaptation in Natural and Artificial Systems", first MIT Press edition, MIT Press: Cambridge, Massachusetts.
- Holland, J. H., 1995, "Hidden Order: How Adaptation Builds Complexity", Addison-Wesley.
- Kauffman, S. A., 1993, "The Origins of Order: self-organization and selection in evolution", Oxford University Press: New York.
- Kittel, C., 1971, "Introduction to Solid State Physics", 4th edition, John Wiley & Sons: New York.
- Lewis, H. R. and Papadimitriou, C. H., 1981, "Elements of the Theory of Computation", Prentice Hall: New Jersey.

- Marijuán, P. C., 1994, "Chapter 5: Enzymes, Automata and Artificial Cells" in "Computing with Biological Metaphors", edited by Paton, R., Chapman and Hall: London.
- Michalwicz, Z., 1996, "Genetic Algorithms + Data Structures = Evolution Programs", Springer-Verlag: Berlin.
- Mitchell, M. and Forrest, S., 1995, "Genetic Algorithms and Artificial Life" in "Artificial Life: an Overview", edited by Langton, C. G., MIT Press.
- Paton, R. C., 1993, "Some Computational Models at the Cellular Level", *BioSystems*, 29, 63-75.
- Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P., 1995, "Numerical Recipes in C: the Art of Scientific Computing", 2nd edition, Cambridge University Press: New York.
- Rosenberg, R. S., 1967. "Simulation of Genetic Populations with Biochemical Properties", PhD Thesis, University of Michigan.
- Rubinow, S. I., 1980. "Ch 6.3 Facilitated Diffusion" in "Mathematical Models in Molecular and Cellular Biology", edited by Segel, L. A., Cambridge University Press: Cambridge.
- Rumelhart, D. E., McClelland, J. L. and the PDP Research Group, 1986, "Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations", MIT Press: Cambridge, Massachusetts.
- Segel, L. A., 1980a. "Ch 1 Biochemical Reaction Theory" in "Mathematical Models in Molecular and Cellular Biology", edited by Segel, L. A., Cambridge University Press: Cambridge.
- Segel, L. A., 1980b. "Ch 6.1 The general balance law and the diffusion equation" in "Mathematical Models in Molecular and Cellular Biology", edited by Segel, L. A., Cambridge University Press: Cambridge.
- Solomon, Berg, Martin, Vilee, 1993, "Biology", 3rd edition, International edition, Saunders College Publishing.
- Stroustrup, B., 1991, "The C++ Programming Language", 2nd edition, Addison-Wesley.
- Wilson, S. W., 1989, "The Genetic Algorithm and Simulated Evolution" in "Artificial Life", ed. Langton, C. G., Addison-Wesley.
- Weinberg, R., 1970. "Computer simulation of a Living Cell", PhD Thesis, University of Michigan.

Zeigler, B. P., 1980. "Ch 2 Simplification of biochemical reaction systems" in "Mathematical Models in Molecular and Cellular Biology", edited by Segel, L. A., Cambridge University Press: Cambridge.