

Library Declaration Form



University of Otago Library

Author's full name and year of birth: ,
(for cataloguing purposes)

Title of thesis: A COSC480 Report:
The Effect of Multiple Populations on Genetic Algorithms

Degree: Postgraduate Diploma in Science

Department: Department of Computer Science

Permanent Address:

I agree that this thesis may be consulted for research and study purposes and that reasonable quotation may be made from it, provided that proper acknowledgement of its use is made.

I consent to this thesis being copied in part or in whole for

i) a library

ii) an individual

at the discretion of the University of Otago.

Signature:

Date:

A COSC480 Report:
The Effect of Multiple
Populations on Genetic
Algorithms

Robin Sheat

Supervisor: Anthony Robins

submitted in partial fulfilment of the
Postgraduate Diploma in Science
at the University of Otago, Dunedin,
New Zealand.

October 2002

Abstract

Genetic algorithms (GAs) are a method of using the concept of ‘survival of the fittest’ to find solutions to some problems. They involve taking a collection of possible solutions to a problem, and gradually changing and combining them until an optimal solution is found.

This report explores a method of reducing the time it takes a GA system to come to a solution by using multiple populations, and migrating the better solutions from each population to the others.

Two methods are introduced and tested: (i) making a copy of the best, and sending them to all the other populations, and (ii) removing the best, and placing them in other populations.

Contents

1	Introduction	1
1.1	Background	1
1.1.1	Terms	1
1.1.2	Genetic Algorithms	2
1.1.3	Multiple Populations	4
1.2	Project Aims	5
1.3	Overview	6
2	Literature Survey	7
2.1	GA History and Background	7
2.2	Multiple Populations	10
3	Implementation	13
3.1	The Library	13
3.1.1	Genome class	14
3.1.2	Cell class	14
3.1.3	Environment class	14
3.1.4	Using The Library	14
3.2	The Application	15
3.2.1	Migration	16
4	An Investigation of Migration	19
4.1	Experimental Method	19
4.1.1	Copy Migration	20
4.1.2	Move Migration	21
4.1.3	The Problem	21
4.2	Results	23
4.2.1	Migration by Copying	23
4.2.2	Migration by Moving	25
5	Discussion	31
5.1	Analysis of Results	31
5.1.1	Migration by Copying	31
5.1.2	Migration by Moving	32
5.2	Further Research	32

A	Initial Aims and Objectives	35
B	Source Code For Library	36
B.1	Environment class (abstract)	37
B.1.1	Constructor	37
B.1.2	initCells (abstract)	38
B.1.3	generation	38
B.1.4	sortCellList	38
B.1.5	fitness (Abstract)	39
B.1.6	crossover	39
B.1.7	random (zero arguments)	40
B.1.8	random (one argument)	40
B.1.9	getNextID	40
B.1.10	getFitnesses	41
B.1.11	getCellConvergence (abstract)	41
B.1.12	getPopnConvergence	41
B.1.13	clearConvergence (abstract)	42
B.1.14	getCosmicRayFreq	42
B.1.15	setCosmicRayFreq	42
B.1.16	getCrossoverRate	42
B.1.17	setCrossoverRate	42
B.1.18	getCell	42
B.1.19	setCell	43
B.2	Cell class (abstract)	44
B.2.1	Constructor	44
B.2.2	mutate (abstract)	45
B.2.3	crossover (abstract)	45
B.2.4	getID	45
B.2.5	setID	45
B.2.6	getParents	45
B.2.7	setParents	45
B.2.8	getGenome	45
B.2.9	setFitness	46
B.2.10	getFitness	46
B.2.11	dirty	46
B.2.12	makeDirty	46
B.2.13	makeClean	46
B.2.14	copy (abstract)	46
B.3	Genome class (abstract)	47
B.3.1	Constructor	47
B.3.2	randomise (abstract)	47
B.3.3	copy (abstract)	47

C	Source Code For Application	48
C.1	Test Program 2	49
C.1.1	main	49
C.1.2	Constructor	49
C.1.3	doCommandLine	50
C.1.4	parseCmdLine	50
C.1.5	loadScript	53
C.1.6	dispatch	53
C.1.7	cmdSet	54
C.1.8	resizePopns	56
C.1.9	cmdCreate	57
C.1.10	cmdDestroy	57
C.1.11	cmdRun	57
C.1.12	cmdExtract	60
C.1.13	cmdStats	61
C.1.14	cmdExtract	62
C.1.15	cmdReplace	62
C.1.16	cmdReset	63
C.1.17	byteToHex	63
C.1.18	toDP	64
C.2	EnvWrapperT2	65
C.3	EnvironmentT2 class	66
C.3.1	Constructor	66
C.3.2	initCells	66
C.3.3	generation	66
C.3.4	fitness	66
C.3.5	getCellConvergence	68
C.3.6	clearCellConvergence	69
C.3.7	random	69
C.4	CellT2 class	70
C.4.1	mutate	70
C.4.2	crossover	70
C.4.3	copy	70
C.4.4	random	71
C.5	GenomeT2	72
C.5.1	Constructor	72
C.5.2	randomise	72
C.5.3	copy	72
C.5.4	getGene (one argument)	73
C.5.5	getGene (no arguments)	73
C.5.6	putGene	73

List of Figures

1.1	The effect as population size increases, averaged (mean) over 100 tests.	5
2.1	Shekel's Foxholes	8
3.1	UML diagram of the library and application	17
4.1	The grid organisms are tested on	22
4.2	Copy migration, two populations, 20 organisms	24
4.3	Copy migration, two populations, 30 organisms	24
4.4	Copy migration, two populations, 40 organisms	25
4.5	Copy migration, three populations, 20 organisms	26
4.6	Copy migration, three populations, 30 organisms	26
4.7	Copy migration, three populations, 40 organisms	27
4.8	Move migration, two populations, 20 organisms	27
4.9	Move migration, two populations, 30 organisms	28
4.10	Move migration, two populations, 40 organisms	28
4.11	Move migration, three populations, 20 organisms	29
4.12	Move migration, three populations, 30 organisms	30
4.13	Move migration, three populations, 40 organisms	30

Chapter 1

Introduction

Genetic algorithm (GA) systems use principles taken from the natural world in order to efficiently search a problem space to find a good solution in that space. The underlying idea behind GA systems is basically one of ‘survival of the fittest’. The search that they do is a heuristically driven one, and is performed by altering a possible solution in ways analogous to biological mutation and genetic recombination. This report details efforts towards optimising this process further, by separating the population into a number of distinct sub-populations, and allowing limited migration between them.

1.1 Background

1.1.1 Terms

The language of genetic algorithms is strongly influenced by biology. Below are terms that are used to describe the elements of the system in this report.

organism A blanket term describing an individual representation of a potential solution to the problem. The most important feature of an organism is the genotype (chromosome) that it contains that is the representation of this particular solution. Information such as ancestry may also be included, depending on what is desired.

gene An individual element, a number of which make up a chromosome. For example, if the solution being sought by the system is a number, then a gene may be a binary digit. Genes are typically what mutation changes.

genotype A sequence of genes. A genotype is a representation of a solution to the problem. This tends to be synonymous with chromosome.

phenotype This is the genotype after it is turned into a potential solution. For example, if the genotype is the binary number ‘1001’, the phenotype may be ‘9’.

crossover This is where the genotypes from two organisms are combined to produce a new one. For example, if genotype *A* is ‘12345’, and genotype *B* is ‘67890’, these may cross to produce a number of different possible results, such as ‘12390’ and ‘67845’.

mutation This is where a random point on a genotype is selected, and changed at random.

population This is simply a group of organisms. Generally, crossover may happen only between organisms within a population.

1.1.2 Genetic Algorithms

A genetic algorithm is based on a collection of representations of potential solutions, i.e. genotypes. There is no particular form that these representations must take, however they are typically designed so that the problem domain can be easily expressed in them. Generally the representations are a binary string, however there are situations where, for example, real numbers may be more appropriate.

A population consists of a number of these, with a size that is generally fixed. Every generation, the genotype from each organism is converted into its phenotype and is checked against a fitness function. The fitness function gives it a value corresponding to how good it is, generally with a higher value representing a better answer. For example, if the purpose of the system was to find the global minima of

$$f(x) = x^2$$

then we could provide

$$f(x) = -x^2$$

as a fitness function, and the system will eventually settle on $x = 0$ as being the best solution.

In order to generate the solutions, the system is started with each organism being randomly constructed. The fitness function is applied to every organism, and the best

ones undergo crossover, with the new organisms created from this replacing the worst. This tends to create a situation, where good pairs combine with one-another, and the best descendants comprise most of the population, and the average fitness of the population rises. It can happen that the random starting state may not contain the required genetic material to get a solution, so the mutation operation is required in order to allow new genes to be created.

With this in mind, the step-by-step operation of a typical GA system is as follows:

1. Create a randomly generated population of genotypes.
2. Arrange the population according to fitness of the phenotypes.
3. Perform crossover on the best ones, with the newly created organisms replacing the worst.
4. Randomly mutate a, typically small, selection of individuals in the population.
A common alternative is to have a chance of mutating the organism produced when crossover occurs.
5. If an acceptable solution has not been generated, go to step 2.

There are a few factors involved here that can affect how the system comes up with a solution. It isn't necessarily most efficient to always take from the very best for crossover. It is better to select at random, but bias in favour of those at the top. This keeps the best ones around, but also allows more genetic diversity which could help prevent the system getting stuck in a local minima. It is also necessary to avoid mutation rates that are too extreme. If the rate is too high, then organisms may get too damaged, and a solution isn't reached. If it is too low, then the system may take too long in coming to a solution.

In some cases, another genetic operator is also introduced: inversion. This is where a segment of the genotype is reversed (Beasley, Bull, and Martin, 1993b). It is not implemented in the system detailed in Chapter 3, as a key idea there is to keep the system relatively simple. Inversion is useful when there is a distinct meaning placed on the order of the genes.

An important part of using a GA system is determining when you have come to a solution. There are two ways of doing this:

1. Has the population reached convergence?
2. What is the maximum or average fitness?

Convergence is a measure of how alike each genotype is compared to the others. A gene is said to have converged when 95% of the population share the same value. The entire population is considered to be converged when all of the genes have converged (Beasley et al. (1993b), cited in De Jong (1975)). The logic behind this is that if the population has converged, then we are in a situation where any deviation from the current best is being punished, and the population can't get any better. There is a problem with this however. If the population is stuck in a local minima, it may take some time and a few lucky mutations to pull it out.

There are some circumstances where you can use your fitness function as a guide. In the $f(x) = -x^2$ example above, we know that the maximum fitness is zero and can stop when we get an organism that reaches that. However in more complex problems, and the sort that GA systems are likely to be used against, it is possible that the maximum fitness isn't known.

1.1.3 Multiple Populations

On their own, GA systems are very effective, however sometimes they can take a long time to come to a solution. One potential way of speeding them up is to look at the idea of multiple populations. This is where there is a number of separate populations of organisms, but with the ability for information, in the form of organisms, to move between them. This can be likened to a geographical situation where you have several populations separated by something like a mountain range, but every so often some individuals manage to travel across, contributing anything useful that evolved in its original home to the gene pool on the other side.

On a typical GA system, simply increasing the population size doesn't consistently decrease the number of generations taken to a solution. The benefit gets less and less, and eventually a point is reached where the computational overhead due to sorting a large number of entries exceeds the gain produced by them coming to a solution faster. Figure 1.1 illustrates the lessening of benefit gained from increasing the population size on the problem described in Section 4.1.3.

As well as giving us the potential of a speedup due to benefits of the method, the use of multiple populations also allows for a very natural parallelisation using multiple computers or processors.

Just like with traditional GA systems, there are a number of parameters that can greatly affect the behaviour of the system. The migration frequency is a measure of how often organisms travel between locations, and the migration rate is how many travel

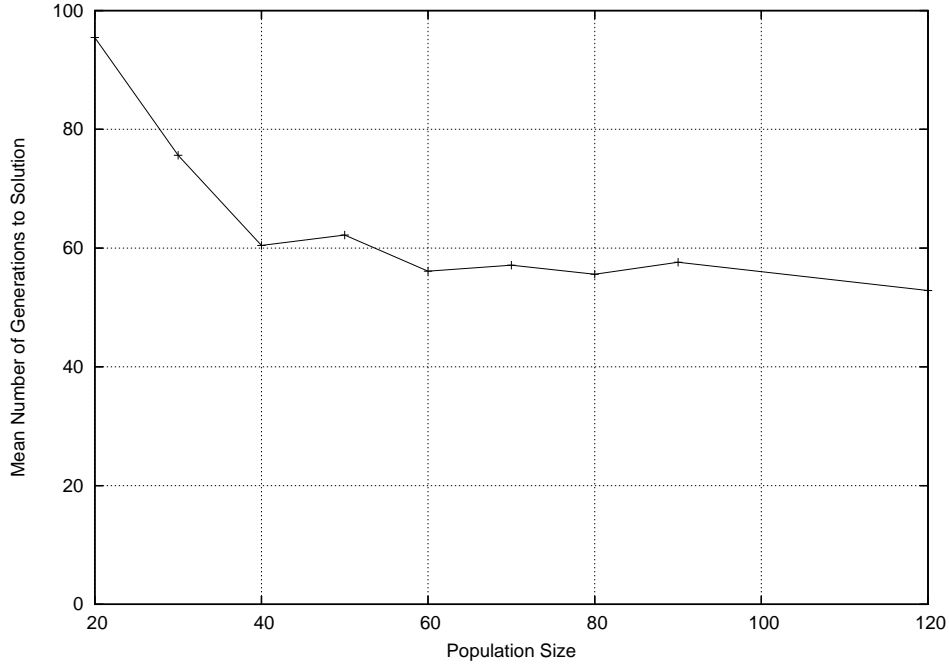


Figure 1.1: The effect as population size increases, averaged (mean) over 100 tests.

each time. There is also the selection and replacement policy, which determine who gets picked to travel, and where they are placed when they get to the destination, if applicable. Other research has shown that the best method for determining migration selection and placement is to select the best from the source, and replace the worst at the destination (Cantú-Paz, 1999a).

Another thing to consider is how the migration happens. There is the biologically inspired method, where a particular organism is removed from its source population and moved to another. Given the ability to copy easily in an electronic medium, we can also send duplicated organisms to the new location. For this report, both methods were implemented and tested.

1.2 Project Aims

The aim of this project was to build a general library that will allow the rapid development of a GA system. With this library, build a system that creates a genetic algorithm environment with the concept of exchange between separate populations, and analyse the effect this has on the evolution of solutions. There are many factors and measures of quality that can be applied to GAs, for the purpose of this the only

one that will be looked at is the time, in generations, to a solution. A breakdown of the initial aims and objectives can be found in Appendix A.

1.3 Overview

In Section 3 I will be talking about the actual implementation of the system, both the library part, and the application developed to do multiple population testing. Section 4 contains the results extracted from the system with various parameters changed in order to see the effect that they have, along with some analysis of them. Finally, Section 5 will draw together all the various results, and provide a determination of what they mean.

Chapter 2

Literature Survey

2.1 GA History and Background

The idea of natural selection incrementally changing something in a way which leads to its gradual improvement has been known for some time. In more recent times, this concept has been applied to instances of solutions to problems.

There has been a great deal of work done on genetic algorithm systems in the past. An influential early work is De Jong's 1975 PhD thesis (De Jong, 1975). De Jong analyses the effects of various parameter changes on GA systems, and details a set of test problems that have been used in many other papers since then. A particularly difficult problem is getting genetic algorithms to find the global minima of the function shown in Figure 2.1.

This set off a large amount of research a while later, with many books and papers coming out in the 1990's, especially, although by no means exclusively, early on. John Koza, a very influential researcher in the field started much of it off with his book in 1992, which contained a broad range of information and example code (Koza, 1992).

Over the next ten years, there was a substantial amount of papers and books produced on the various subtopics of genetic algorithms and related topics. Many ones relevant to this report tend to come out of the Illinois Genetic Algorithms Laboratory (IlligAL)¹.

The general GA principles used in the system described in this report are based on those detailed in Beasley, Bull, and Martin (1993a), with some simplifications and alterations. This paper details what is pretty much the canonical GA system. The main points of divergence between what is described in Beasley et al. (1993a) and my

¹<http://www-illigal.ge.uiuc.edu/index.php3>

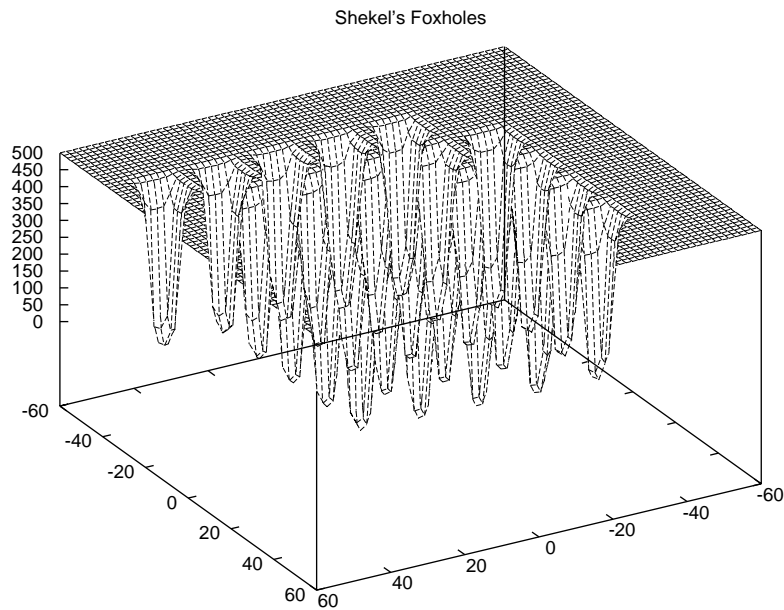


Figure 2.1: Shekel's Foxholes

implementation are in the mutation and the crossover selection. Beasley et al. describe tournament crossover selection, where a probability of being selected is assigned to each organism according to its fitness level. My implementation does straight selection of the best. The paper also describes mutation as being done to the organisms resulting from crossover as they are created, in my version the individual to mutate is selected randomly.

The crossover method that the paper explains, single-point crossover, is the same as the one implemented in this project.

As well as describing the standard GA system, Beasley et al. (1993a) also looks at a few other variations, such as different search methods. After this, it goes into a description of the theory behind GAs, and why they work. An interesting point mentioned here is that:

There is no accepted “general theory” which explains exactly why GAs have the properties they do. Nevertheless, several hypotheses have been put forward which can partially explain the success of GAs. These can be used to help us implement good GA applications. (Beasley et al., 1993a, Section 4.1)

The authors then proceed to discuss such hypotheses as schemata (Holland, 1975),

the building block hypothesis (Goldberg, 1989), and principles such as exploration and exploitation.

Schemata, and schema theory is a description of how genes that are ‘good’ (contribute to a high fitness score) tend to remain in the population and end up combining with others, which causes the fitness of these combinations to increase. If there is a sufficient variety of schema in the population, then many places in the search space are being tested. This is known as implicit parallelism.

The building block hypothesis explains the effectiveness of GAs as a collection of genotypes that gradually ‘pick up’ good genes. This relies on the coding scheme to an extent. Beasley et al. (1993a), Section 4.2 states that the important aspects of a coding scheme are that:

1. any genes who have a different effect when in the presence of specific other genes should be as close to those other genes as possible, while
2. there is as few as possible incidences of such interaction between genes.

If there is a pair of genes, both of which are required in order to get any benefit from either of them, adhering as close as possible to these rules reduces the chance of them being separated during crossover.

The concepts of exploration and exploitation described by Beasley et al. (1993a) is of particular relevance to this report. With a finite population size, some trade-off must be made between concentrating the search around a known-good location on the fitness-space, and looking in other locations for somewhere that is hopefully even better. In a typical GA system, mutation is the operator that contributes to exploration, while crossover does exploitation. In a multiple population system, the separation of the populations allows each population to perform its own exploration/exploitation process with ‘refined’ suggestions for exploration sent to the other populations on a regular basis.

The paper proceeds to discuss other things that are important in the context of GAs, but less important for this report, such as optimising of fitness functions, and applications of GAs in the real world.

One other thing that is mentioned of relevance however is two problems that GA systems are vulnerable to. These were briefly mentioned in Section 1.1.2. They are premature convergence and slow finishing.

Premature convergence is caused when a few individuals that are fit compared to the rest of the population, but aren’t optimal, end up filling the population, which

causes it to converge on a local minima. Once this has happened, the only way out is with mutation, which is equivalent to a slow random search. Beasley et al. (1993a) detail a solution which involves controlling the reproduction of organisms.

Slow finishing is effectively the opposite problem. It is caused when the population has mostly converged, but hasn't yet located the global minima. There is little difference between the best individuals and the average ones, and so there isn't a large amount of selection pressure. If fitness scaling is used, which is sometimes necessary when a proportional selection method is used, then a particularly poor organism can cause this to occur. This may also be a sign of having a too high mutation rate.

2.2 Multiple Populations

Multiple populations in GAs comes under the wing of a somewhat broader concept of parallel GAs. Sprave (1994) refers to the main types of these as the *migration model* and the *diffusion model*. The former is the one relevant to this report.

Erick Cantú-Paz of IlliGAL has done a significant amount of work towards developing the field of multiple population genetic algorithms. In Cantú-Paz (1999a), the various methods of selection and replacement are described and compared. The four methods discussed are:

1. Select good, replace random.
2. Select good, replace bad.
3. Select random, replace random.
4. Select random, replace bad.

Initially, the comparison of methods is approached from a theoretical basis, looking at the proportion of good individuals with every generation under the different methods. The result of these is that the 'select random, replace random' method is equivalent to a single population, and that the best method is 'select good, replace bad'. Experimental results are also used to confirm the effects.

The claim that multiple populations lead to superlinear speedups is looked at, and it is determined that in many cases the speedup is gained by the increased selection pressure from selecting the good individuals, and replacing the worst ones. However, the point is made that this may not always be the case. There may be circumstances in which real-time speedup is gained by such things as multiple populations being able

to fit into the processor's cache better. These types of considerations are not looked at in this report.

Another paper by the same author, Cantú-Paz (1999b), discusses a few other parameters of multiple population GA systems. In short, it looks into optimising the parameters for degree of connection of each population (how many neighbours each population has), and the rates of migration. These results are not entirely relevant to this report, as they have a significantly different migration model. In Cantú-Paz (1999b), all the populations evolve to convergence, and then migration occurs, and the cycle repeats. Other studies have shown that this can give equal results to the method described in this report, which has regular migration regardless of the convergence level.

A comparison of various different methods of doing migration is done by Wang, Maciejewski, Siegel, and Roychowdhury (1998). The problem domain here is the travelling salesman problem. Five methods are compared:

independent Each population evolves separately, and there is no exchange between them at all. This is equivalent to running a single population system many times, and taking the best result.

migration This is closest to the copy migration done in this report. It has a number of populations, and a copies of the best organisms migrate between them regularly. There are a few differences between Wang et al.'s procedure and the one in this report. These are detailed below.

partitioned The search space is split into a number of partitions, each population only searches its part.

segmentation The genotype starts off as a small part of a complete potential solution, and as good ones are found, they are combined to create larger solutions, which are further combined until a complete, good solution is found. Note that no migration happens in this, however the results of the populations are combined when the larger parts are created.

segmentation-migration This is very similar to segmentation, however there are periodic migrations.

The last two are designed in a way that is quite domain specific, but could probably still be adapted to other problems.

The differences between the system described in Chapter 3 and Wang et al. (1998) are substantial, but are likely not sufficient to make any comparison invalid. In Wang et al. (1998), there are a larger number of populations, typically sixteen. The migration between them is done by making a copy of the best one from each population P_i , and sending it to $P_{(i+1) \bmod N}$ for the first migration event, $P_{(i+2) \bmod N}$ for the second, and so on (where N is the number of populations). The results that they get are encouraging, and show that the migration method typically does very well.

Chapter 3

Implementation

The GA system was written in Java, as its object-oriented nature allows for the code to be written in a modular way, allowing different methods to be experimented with easily. There are other practical benefits, such as platform independence and fast development time.

There are two main parts to the implementation:

1. a GA library, providing functions and a framework that can be used by a programmer wishing to implement a GA system (Section 3.1), and
2. an implementation of a complete GA system, used to generate the results in this report (Section 3.2).

3.1 The Library

The library is a set of three abstract classes. They allow data to be passed around internally in a consistent way, even though the specifics of things such like the storage of the genotype may vary between implementations.

Each class must be extended in order to provide functionality specific to the problem that will be applied to the GA system. The classes are:

Genome This class contains the genotype,

Cell This class contains a **Genome** instance, and the other information such as caching the fitness to avoid the need for recalculation, and family history. This contains all the information that would be associated with the term ‘organism’ described in Section 1.1.1.

Environment This class contains a collection of **Cells**, the fitness function, and governs the interactions between the organisms it contains.

3.1.1 Genome class

This class allows for common storage of the genotype. This is necessary as the actual stored format of the genotype will vary according to the problem domain. This class doesn't provide actual storage, but it does define a few methods that can be used in order for other classes (such as the **Cell** class) to perform common operations on the genotype, no matter what the representation is.

3.1.2 Cell class

The **Cell** class provides an abstraction of the genotype. It allows the genotype to be dealt with indirectly, and contains extra information such as the fitness of the cell. It includes calls to do such actions as crossover and mutation which are implemented by the extending class. There is also the facility for unique identification strings, and keeping family trees, however these haven't been fully implemented due to the lack of need for this project.

3.1.3 Environment class

The **Environment** class does most of the work out of the library classes. It controls the specific actions that happen in the environment, such as determining when things like crossover and mutation happen, and ensuring that they do. As most of the low level processing has been put into the **Cell** and **Genome** classes, the abstract version of the **Environment** class is fairly complete. The main thing that an implementing class needs to define is the fitness function.

3.1.4 Using The Library

The classes detailed above provide an abstract model of a complete GA system. There are other factors such as a user interface, and for the purpose of migration, something to move **Cells** between **Environments**. However, most of the work related to the individuals in the system is done in these. The application class that does the administration of the program needs to call the *generation()* method in the **Environment** class. This goes through the following steps:

1. Ensure that the **Cells** are sorted into fitness order to aid later operations that depend on the ranking of the cells
2. Perform crossover on the population:
 - (a) Select a number of individuals from the best of the population, equal to twice the number of new ones we want to generate.
 - (b) Select random pairs from the above selection, and call the *crossover()* method in the **Cell** class on them.
 - (c) Replace the worst individual in the population with this newly created one.
 - (d) Repeat from (b) until we have sufficient new individuals.
3. For each **Cell** in the population:
 - (a) Select a random number n .
 - (b) If n is less than or equal to the mutation rate, then call the *mutate()* method on the **Cell**.
4. Clear some internal state information so it will be recalculated if necessary.

The actual implementation of the *crossover()* and *mutate()* methods is left up to the extending class, as it is potentially quite specific to the problem. If the genotype was a tree structure, then quite different operations would be necessary than if it was a binary string.

Some of the processes are implemented in ways that are known not to be the best. As an example, the tournament crossover selection methods as described in Beasley et al. (1993a) are better than the method that is performed in the **Environment** class. This was done intentionally with the aim of helping the system as little as possible, so that hopefully any effects are caused only by migration, and not by its interaction with other processes. For an analysis of more efficient methods, see Bäck and Schwefel (1993).

3.2 The Application

The library as described above requires some further code to extend it in order for it to be useful. This section describes the implementation used to generate the results in Chapter 4.

Once a complete single population system was written and working, it was duplicated, and the duplicate was renamed from the original ‘**Test1**’ to ‘**Test2**’, hence the “T2” suffix to the classnames. **Test2** was then enhanced to allow multiple populations, and it was on this program that testing was done.

Most of the classes in this program extend the library classes, and add to them the code required to deal with the specific problem. The classes are:

GenomeT2 This class deals with the specifics of storing and accessing the genotype as a sequence of bytes.

CellT2 This implements the genotype-specific bits, such as crossover and mutation.

EnvironmentT2 This contains functions that compare cells, such as the fitness function and the convergence calculation.

EnvWrapperT2 This holds a particular instance of an **EnvironmentT2** class along with various variables specific to it.

Test2 This handles all the input/output, including user interaction. It also controls the migration between environments.

The interface allows all the variables, such as mutation rate, crossover rate, fitness caps and so on to be specified interactively. It also allows scripts to be executed to allow multiple trials to be run. All the code for this is inside the **Test2** class. This class holds a list of **EnvWrapperT2** instances, each of which holds an **EnvironmentT2** class and associated variables.

Figure 3.1 is a UML diagram of the classes and relationships between them for the application, and the library. The library part is the three abstract classes listed to the left. This diagram only shows the important data fields and methods, there were a number that were only used internally that aren’t included.

There are other features, such as the ability to stop a run when the convergence measure or the average fitness reach a certain value.

3.2.1 Migration

As mentioned above, the **Test2** handles migration. It implements the two different methods mentioned in Section 1.1.3, one involving duplication, where an organism is copied and that copy is placed in the other populations, and the other method where the organism is removed from its initial population and placed in the new one.

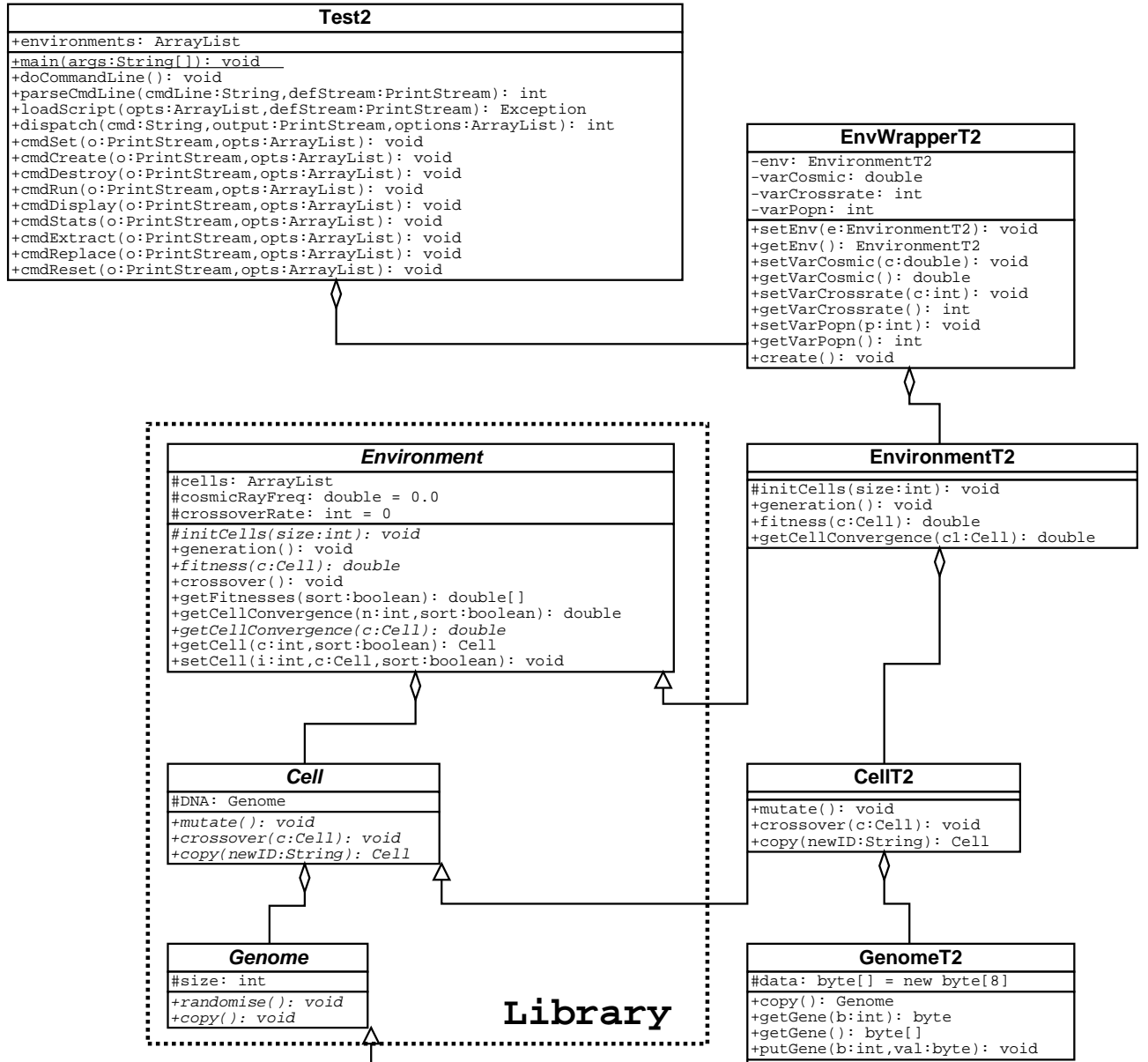


Figure 3.1: UML diagram of the library and application

The algorithm for determining how migrants move in the latter method is:

1. Build a list of all the organisms that are going to migrate.
2. Set a *migrant* pointer to point to the first organism in the list.
3. Set a *destination* pointer to point to the first population.
4. If the organism pointed to by *migrant* is not from the population specified by *destination*:
 - (a) Move the organism from the original population to this one.
 - (b) Increment the *migrant* pointer
5. Increment the *destination* pointer, starting again at the first population if necessary.
6. If there are still organisms to move, go to step 4.

Chapter 4

An Investigation of Migration

4.1 Experimental Method

This section details the actual experimental setup used to generate the results. The task that the GAs were given is described in 4.1.3.

The two methods of migration explored, one where the migrants are copied, and one where they are actually moved, have significantly different results, and so are treated independently.

In these results, there are a number of different factors being considered, which are outlined here.

migration rate Migration rate is a count of how many organisms migrate every time a migration event occurs.

migration frequency This determines the time interval, in generations, between each migration event. A higher value represents less migration. Note that this is quite distinct from migration rate.

number of populations This is how many populations there are in the system, with migration occurring between them.

population size This is a count of how many organisms are in each population.

time to solution This is a measure of the number of generations that have elapsed before the first optimal solution was created.

On the graphs presented here, the independent variable is migration frequency, and the dependant variable is the time to a solution. A generation is a complete crossover

and mutation cycle. All of the tests done here are averaged over 100 samples as there is a large amount of stochastic variation involved. Even with this large number of trials, there is a noticeable amount of variation in the results.

The various graphs illustrate the effects that changing population sizes have on the speed of the GA system. They are all compared to a baseline situation which is a single population with the same number of organisms as the multiple population system. Except for the baseline, each line on the graph represents a different migration rate.

Due to the fact that smaller differences in migration frequency have a greater effect when the frequency is low than when it is high, more sampling was done when the migration frequency was less than ten. This can be seen on the graphs as the closer group of points on the left hand side of each plot.

In these tests, the crossover rate for the multiple populations is one (meaning that one new individual is created for every generation per population), and the mutation rate is 0.05 (meaning that every organism has a one in twenty chance of mutating each generation).

In order to make the comparison between the performance of the single population, and that of the multiple populations, the crossover rate of the single populations is adjusted. For example, if we are comparing a three population system with a crossover rate of one to a single population, we need to ensure that the same number of organisms are created every generation. The three population system will gain three new organisms, one for each population. To compensate for this, we need to increase the crossover rate of the single population to be three. Especially on smaller populations, the crossover rate has a very large effect on the time to a solution.

4.1.1 Copy Migration

For the copy method of migration, each time migration happens, the best organism from each population is copied out, and replaces the worst in all the other populations. This activity happens in parallel, so that the new organisms being inserted don't interfere in the selection process.

The migration rate must be watched in this method, as the number of new organisms a population gets is equal to the number of other populations multiplied by the migration rate. This can easily lead to an attempt to add more organisms to a population than it can hold, or replacing its entire population with new ones.

4.1.2 Move Migration

In this method of migration, the organisms aren't copied. Rather, they are removed from their initial population and placed into the new one. The actual method of determining where the migrating organisms end up is detailed in Section 3.2.1.

When migration happens, the best from each population travel to another population, filling the gap that was newly created there. If they are good enough, they are included in crossover, and so their genetic material gets included in the new population.

4.1.3 The Problem

GA systems can be applied to many different types of problem. For the analysis done in this report, the problem domain is described below.

The task of the GA system is to come up with a genotype consisting of a set of **if...then** statements that will navigate to the centre of a grid as in Figure 4.1. The genotype is evaluated into eight of these conditionals, four of which allow movement in the x direction, four in the y direction. The genotype is dealt with as a sequence of eight hexadecimal bytes, each byte of which represents a complete **if...then** statement. The first nybble is the condition, the second is what happens if the condition is true. The mapping between the nybbles and what they represent is:

Value	1st Nybble (Condition)	2nd Nybble (Consequent)
0-4	< 0	-1
5-9	> 0	+1
A-F	$= 0$	no change

When the phenotype is evaluated, the first part is applied to the x dimension, the second to y , the third to x and so on. For example, the following genotype:

26 27 80 70 05 72 18 17

is evaluated as:

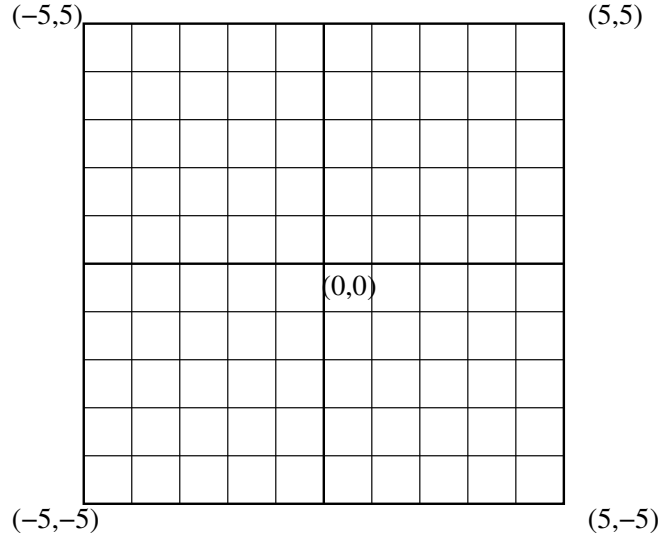


Figure 4.1: The grid organisms are tested on

Genotype part	Phenotype part
26	if $x < 0$ then $x = x + 1$
27	if $y < 0$ then $y = y + 1$
80	if $x > 0$ then $x = x - 1$
70	if $y > 0$ then $y = y - 1$
05	if $x < 0$ then $x = x + 1$
72	if $y > 0$ then $y = y - 1$
18	if $x < 0$ then $x = x + 1$
17	if $y < 0$ then $y = y + 1$

When this is tested, the evaluation works like this, assuming it is started in corner $(-5,5)$:

1. **if** $x < 0$ **then** $x = x + 1$, moves to $(-4,5)$
2. **if** $y < 0$ **then** $y = y + 1$, no change
3. **if** $x > 0$ **then** $x = x - 1$, no change
4. **if** $y > 0$ **then** $y = y - 1$, moves to $(-4,4)$
5. **if** $x < 0$ **then** $x = x + 1$, moves to $(-3,4)$
6. **if** $y > 0$ **then** $y = y - 1$, moves to $(-3,3)$

7. **if** $x < 0$ **then** $x = x + 1$, moves to $(-2,3)$

8. **if** $y < 0$ **then** $y = y + 1$, no change

The fitness value is gained by taking the difference in Manhattan distances from the start and the end. In this example, the initial distance is 10 ($|-5| + |5| = 10$), and the final distance is 5, so the fitness for this is 5. In order to get the final value, this is repeated three more times, once at each corner, and each distance is summed which produces the final score. Due to this one doing better from one side than from the other as a result of it having more $x = x + 1$ statements than $x = x - 1$ statements, it will score five from two tests, and three from the other two, giving it a resulting score of 16, the maximum possible.

4.2 Results

In this section are the experimental results of the system described above.

4.2.1 Migration by Copying

Two Populations Figures 4.2, 4.3, and 4.4 all show the performance of two populations, with the migration rate ranging from zero to three. A migration rate of zero is the situation when there is no migration, so effectively this is the performance of a number of completely independent populations. The line representing a migration rate of zero is quite significantly removed from the other ones, which indicates that there is a very strong effect from the migration. In all the cases, so long as there is migration happening, then there is very little difference between the migration rates. However the tests with a rate of two or three typically did better than the test with a rate of one.

When the number of organisms in each population of the multiple population systems is twenty or thirty, and the migration frequency is set to approximately five, the performance is similar to that of one single population of size forty. However, when the number of organisms in the each population of the multiple population systems is forty, then having a migration frequency of around ten gives it similar performance to a single population of size sixty.

Three Populations When comparing between two and three populations, three scores significantly better, as shown in Figures 4.5, 4.6, and 4.7. However with the

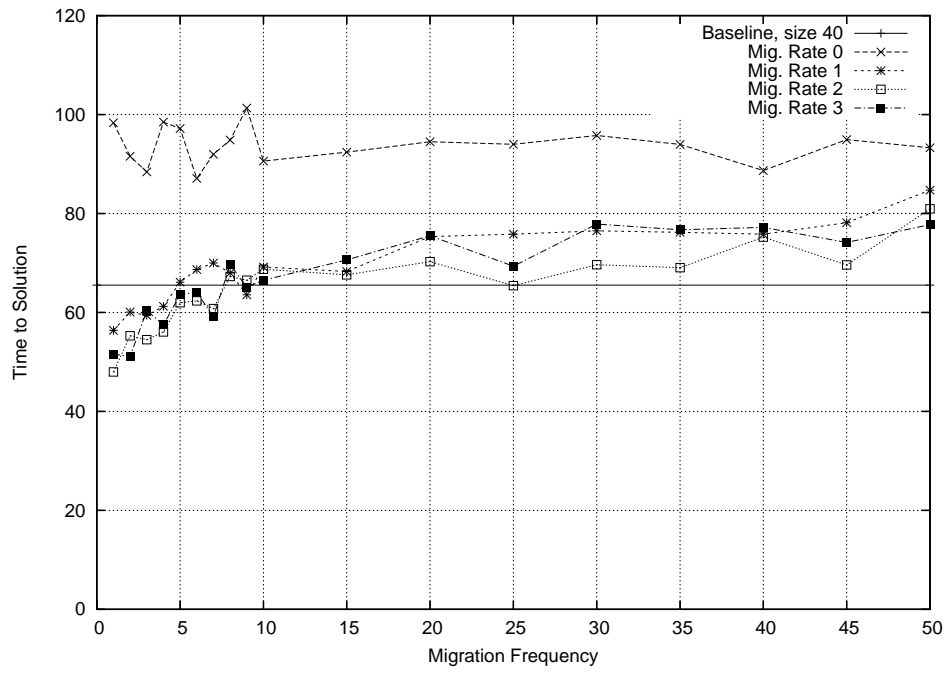


Figure 4.2: Copy migration, two populations, 20 organisms

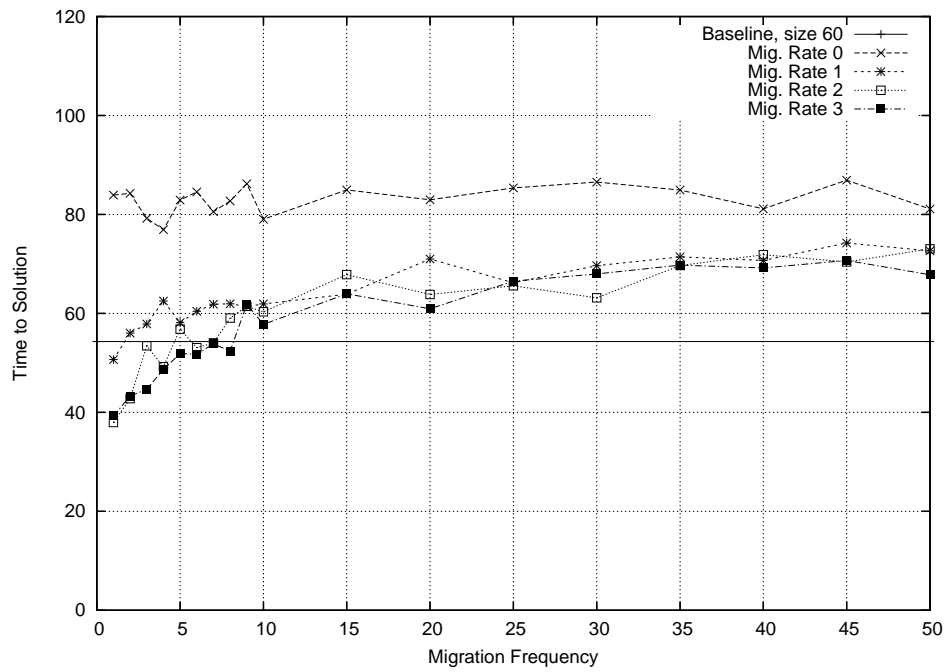


Figure 4.3: Copy migration, two populations, 30 organisms

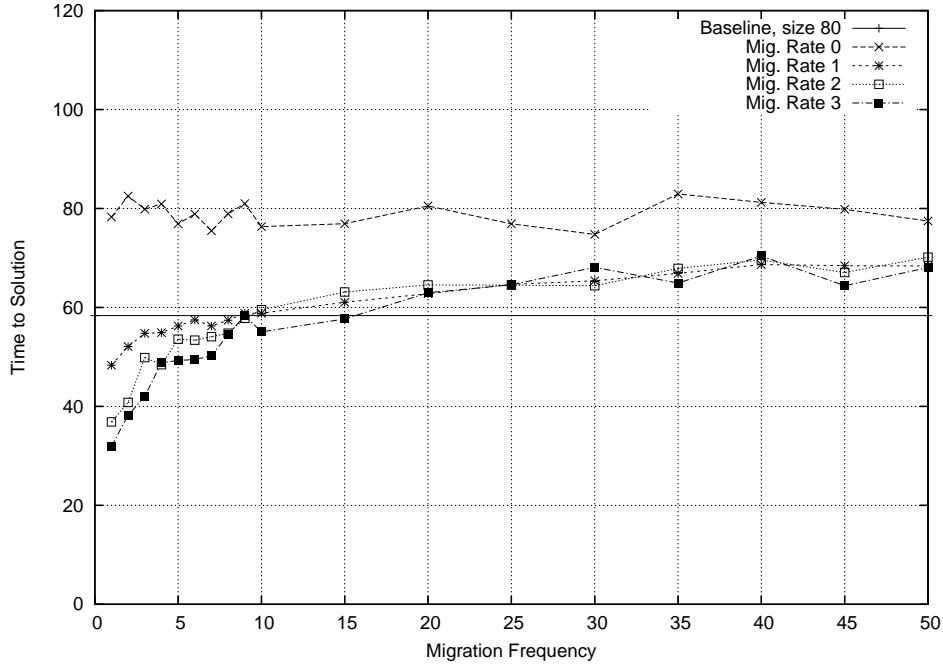


Figure 4.4: Copy migration, two populations, 40 organisms

increased crossover rate giving the single population an advantage, they perform very similarly. About the only difference is that the point where the single population and multiple population systems match times is close to five in all cases.

4.2.2 Migration by Moving

This method never manages to exceed the performance of the baseline single population when the crossover rate is adjusted to keep it equivalent. However, it does do better than a single population without the crossover rate being raised. (Section 4.1)

Two Populations As can be seen in Figures 4.8, 4.9 and 4.10, this method of migration tends to do significantly worse than a single population, although it still does better than a multiple population system with no migration at all (having a migration rate of zero).

It is interesting to note that here also the migration rate makes very little difference, provided it actually happens.

Three Populations The performance of this is similar to that of two populations, however over the three graphs, there is a tendency to get closer to the performance of

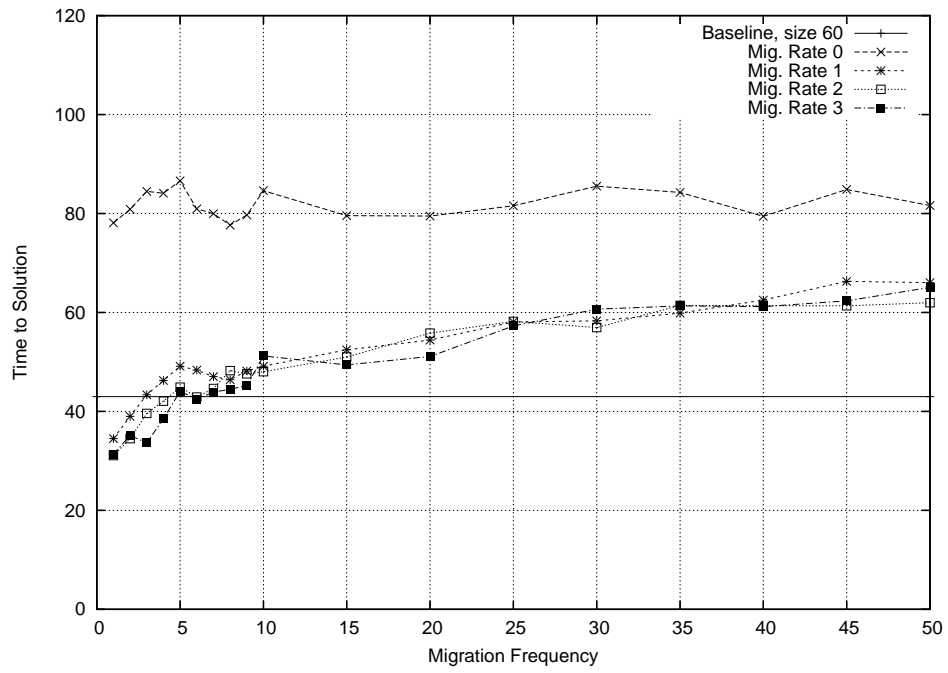


Figure 4.5: Copy migration, three populations, 20 organisms

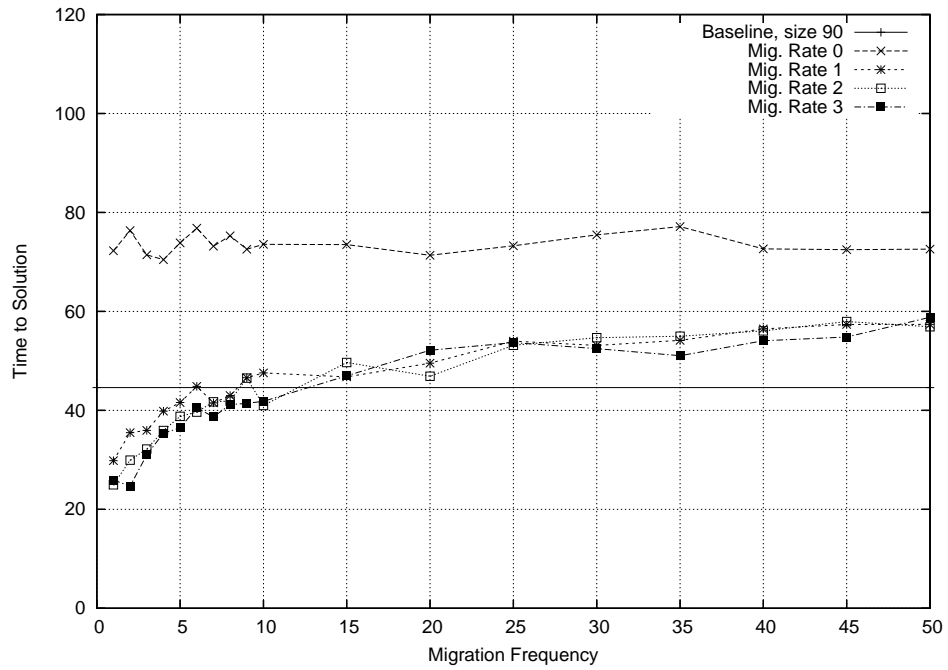


Figure 4.6: Copy migration, three populations, 30 organisms

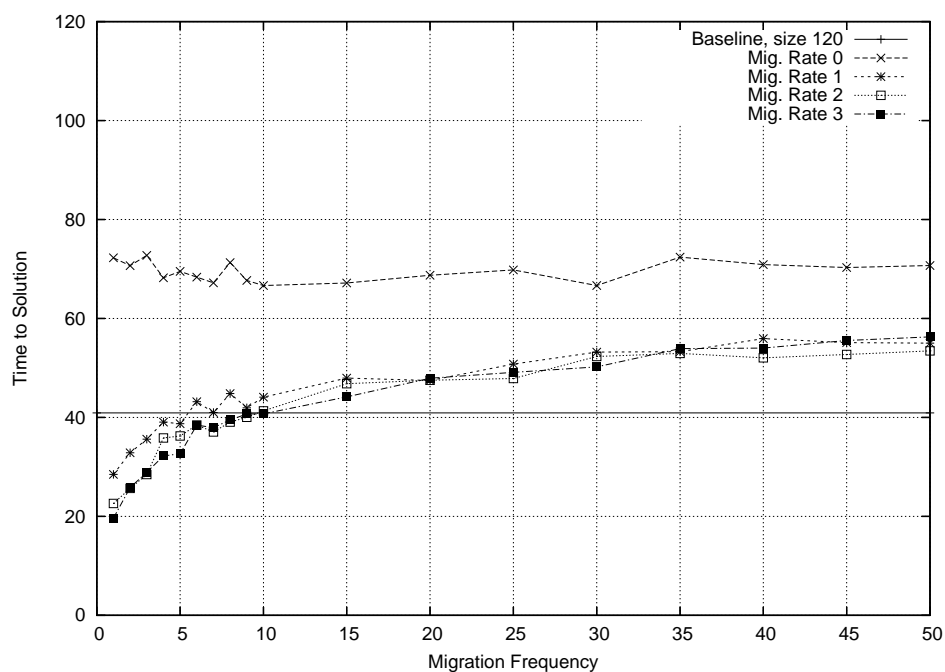


Figure 4.7: Copy migration, three populations, 40 organisms

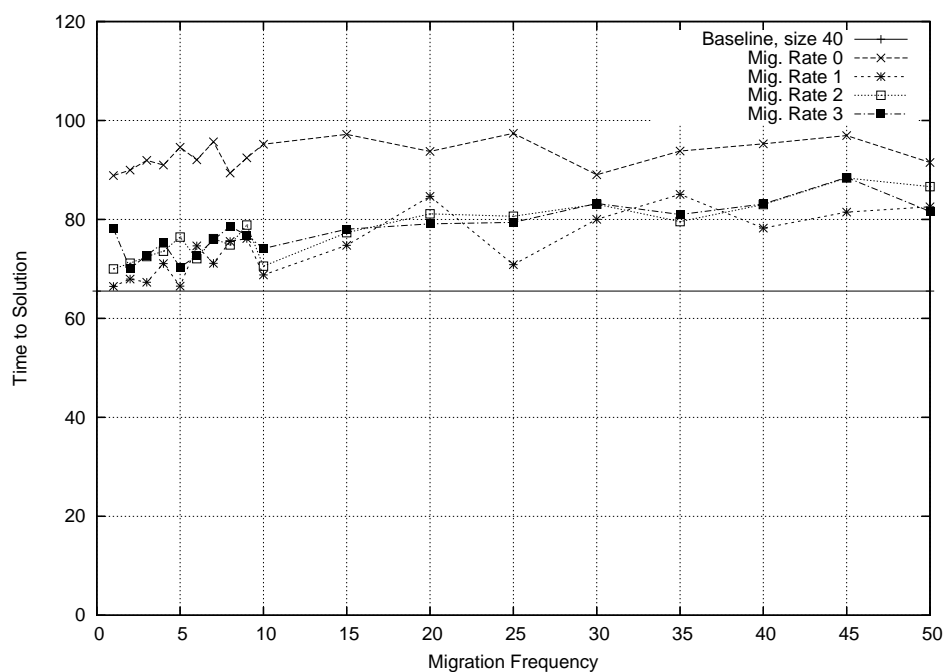


Figure 4.8: Move migration, two populations, 20 organisms

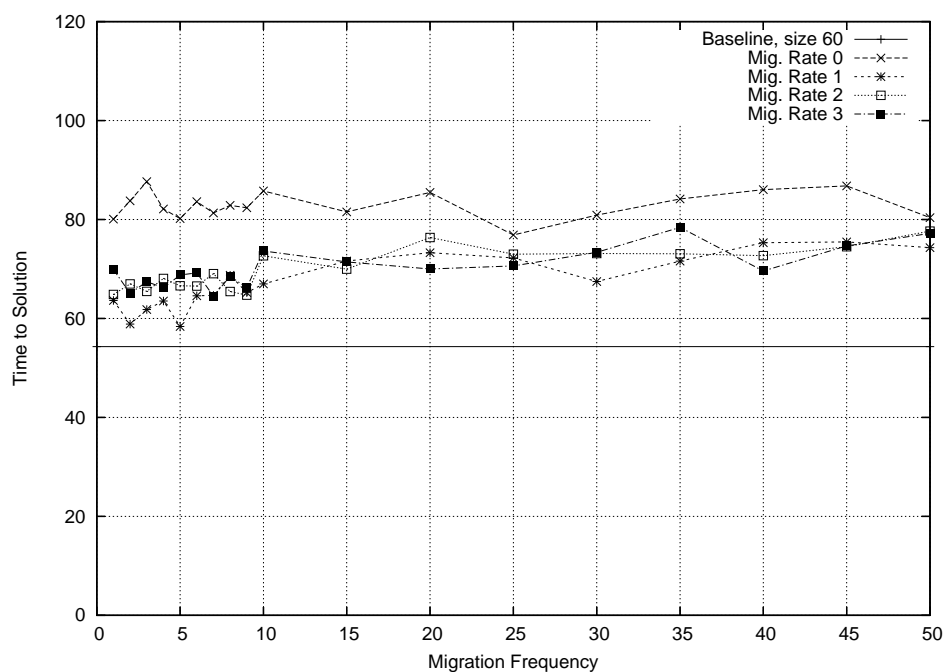


Figure 4.9: Move migration, two populations, 30 organisms

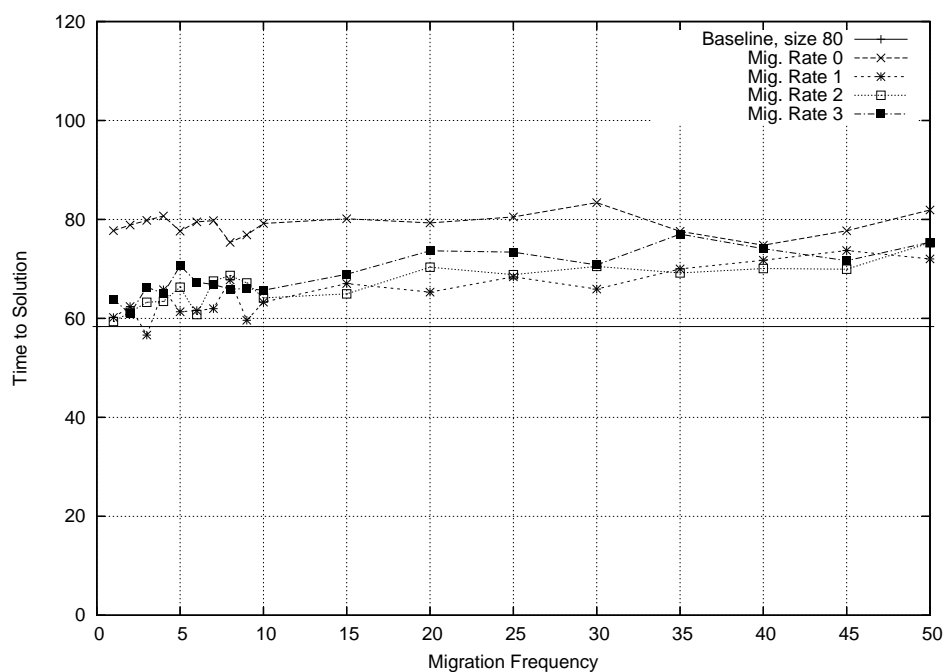


Figure 4.10: Move migration, two populations, 40 organisms

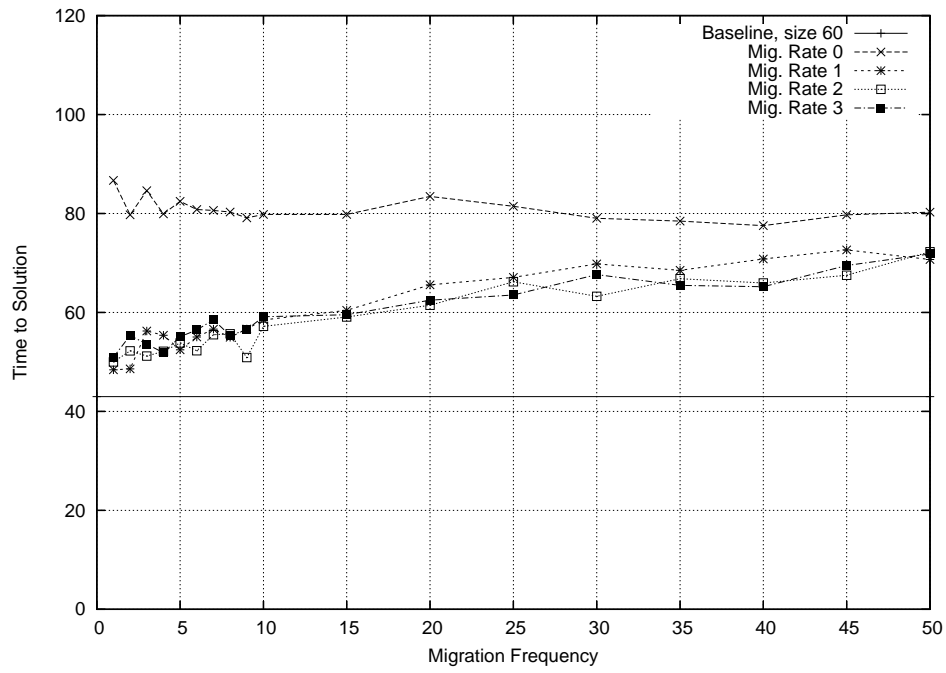


Figure 4.11: Move migration, three populations, 20 organisms

the larger population, as shown in Figures 4.11, 4.12, and 4.13.

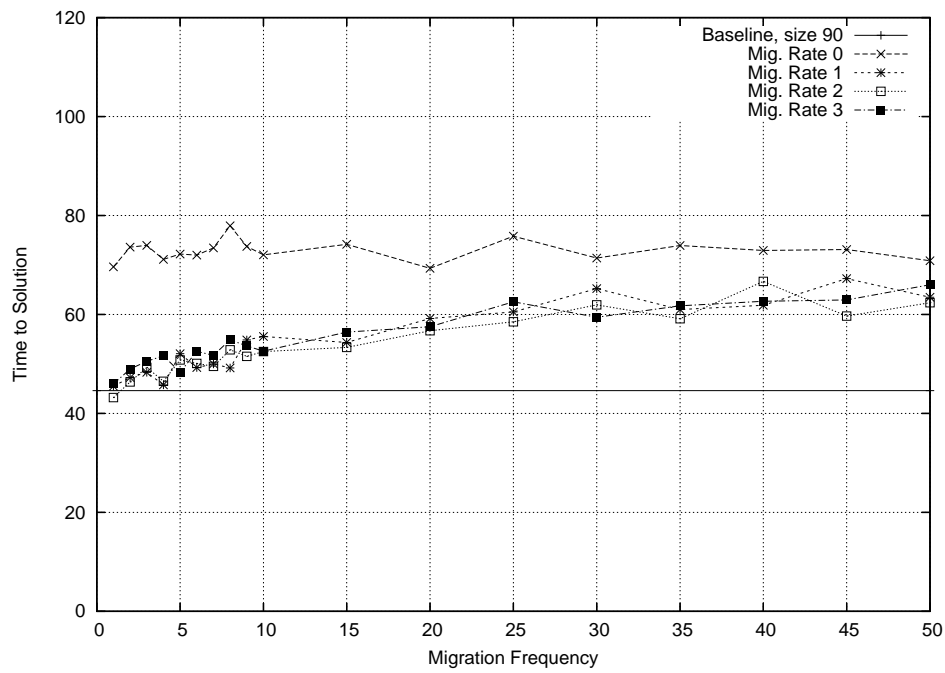


Figure 4.12: Move migration, three populations, 30 organisms

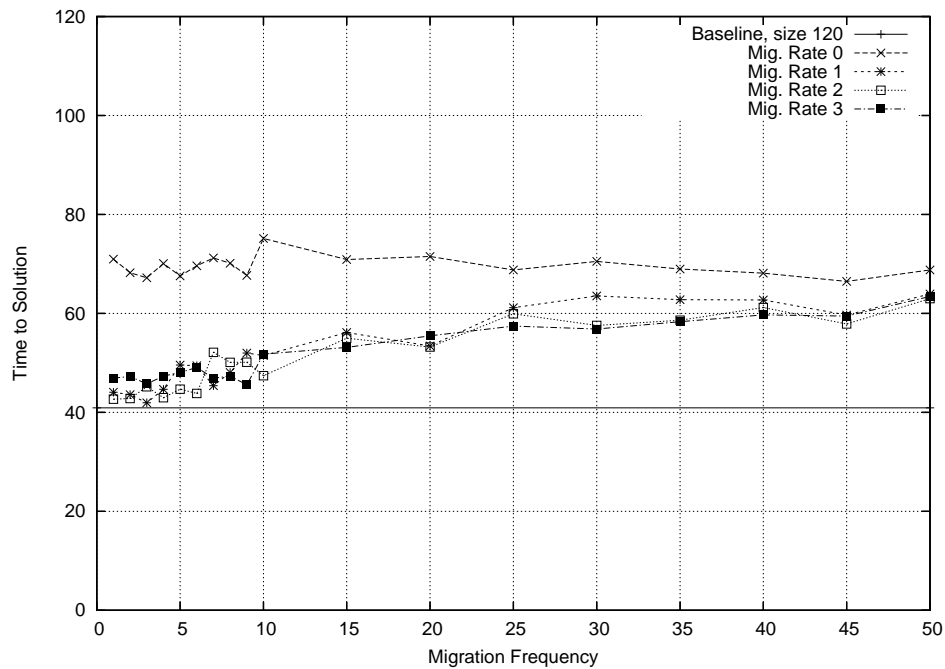


Figure 4.13: Move migration, three populations, 40 organisms

Chapter 5

Discussion

5.1 Analysis of Results

As can be seen from the results, adding migration does have a noticeable effect, when compared to the case of a number of completely separate multiple populations. The general things that can be observed from the results are that migration should happen quite often for the greatest benefit. This is useful in situations where the problem being solved by the GA is quite large, and needs to be spread across a network of computers. Having the network act like one large population requires a large amount of data to be exchanged, which may become a bottleneck. If each computer was a population, and only a few organisms travelled across, then the bandwidth required would be significantly less.

5.1.1 Migration by Copying

With lower delays between migration events, this method performs quite well. The likely cause of its significant edge over the other migration method is the fact that good individuals are actually being copied and are displacing the less good ones, which introduces more selection pressure. While this does make it perform better in this particular problem domain, this also may cause a similar problem to that of having a too high crossover value, i.e. premature convergence. The system has an increased likelihood of getting stuck in a local minima, and not find the optimal solution.

Wang et al. (1998) compares a number of different methods of doing parallel GAs, including one that is very similar to the migration by copying used in this report. The parameters they have are significantly different, however. They work with a migration frequency of five, a migration rate of one, and a very different way of determining

what population to place the migrants in. Despite these differences, many of which would harm the speed to a solution (but have benefits in other areas, such as improving accuracy), it confirms the results found here.

5.1.2 Migration by Moving

This one doesn't do as well as the single large population does, however it does still beat multiple populations with no migration. The reason for the comparatively poor performance is likely to be a result of the fact that, aside from standard crossover, no organisms are artificially created, unlike the other migration method. This will also have the effect that systems using this migration method will be less susceptible to premature convergence, and should give more accurate solutions, as described in Section 2.1.

5.2 Further Research

There are a large number of things that can be changed in this system in order to change the results, only a few of which have been touched on here. Some further things that could be looked at are larger numbers of populations, the crossover rates and the linkages between populations, i.e. if there is a difference between fully connected populations and ones that are not. The tests could also be repeated with a more complex problem, and looked at with an eye to see what the accuracy of the solution at convergence is, and how migration affects that.

There is in many cases a significant amount of interaction between the various parameters of the GA system, and exploring how these affect one another could lead to exploiting particularly efficient combinations.

Another potential avenue for exploration is to determine if varying the fitness function between populations has an effect. As an example, in the sample problem explored in this report, if one population put a greater selection pressure on x and the other put greater selection pressure on y , it is possible that each one would cause part of the solution to be found very fast, which would then be combined to form a complete solution. A couple of different methods of doing this are looked at in Wang et al. (1998). A further variation on these is to have different mutation and crossover rates in different populations, allowing some populations to lightly cover the search space finding viable areas, and others to refine the solutions they encounter to find what the optimal solution is.

References

- Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1(1):1–23, 1993. URL <ftp://ftp.cs.wayne.edu/pub/EC/ES/papers/sys93.ps.gz>.
- David Beasley, David R. Bull, and Ralph R. Martin. An overview of genetic algorithms: Part 1, fundamentals. *University Computing*, 14(2):58–69, 1993a.
- David Beasley, David R. Bull, and Ralph R. Martin. An overview of genetic algorithms: Part 2, research topics. *University Computing*, 15(4):170–181, 1993b. URL <citeseer.nj.nec.com/article/beasley93overview.html>.
- Erick Cantú-Paz. Migration policies, selection pressure, and parallel evolutionary algorithms. Technical Report IlliGAL 99015, University of Illinois, 1999a. URL <citeseer.nj.nec.com/506257.html>.
- Erick Cantú-Paz. Topologies, migration rates, and multi-population parallel genetic algorithms. Technical Report IlliGAL 99007, University of Illinois, 1999b.
- Kenneth De Jong. *The Analysis and behaviour of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan, 1975.
- D.E. Goldberg. *Genetic Algorithms in search, optimization and machine learning*. Addison-Wesley, 1989.
- J.H. Holland. *Adaption in Natural and Artificial Systems*. MIT Press, 1975.
- John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.
- Joachim Sprave. Linear neighborhood evolution strategy. In Anthony V. Sebald and Lawrence J. Fogel, editors, *Proceedings of the Third Annual on Evolutionary Programming*, pages 42–51. World Scientific, 1994.

Lee Wang, Anthony A. Maciejewski, Howard Jay Siegel, and Vwani P. Roychowdhury.
A comparative study of five parallel genetic algorithms using the traveling sales-
man problem. In *IPPS: 11th International Parallel Processing Symposium*. IEEE
Computer Society Press, 1998. URL citeseer.nj.nec.com/318940.html.

Appendix A

Initial Aims and Objectives

Aims

To implement a system of classes or libraries that will allow the evolution and competition of genetic algorithms, and to use this to build a system that creates a genetic algorithm environment that has a concept of physical separation, and analyse the effect this has on the evolution of solutions.

Objectives

- Build a system that evolves genetic algorithms
 - Design a representation of general genetic algorithms that will allow them to be easily manipulated.
 - Design a class or library structure that allows them to be dealt with sensibly.
 - Write the code that does this, and test it on basic cases.
- Explore the effects of geographical separation on the evolution of the genetic algorithms
 - Construct a set of problem domains to be used for testing.
 - Build a program that contains two (or possibly more) “environments” for genetic algorithms to evolve inside.
 - Run tests, varying some parameters between them (along with having a control), and recording results.
 - Analyse results, and determine if there was or was not a significant effect.

Appendix B

Source Code For Library

B.1 Environment class (abstract)

The environment class sets up a structure for cells to be stored in, and compares them to each other to determine fitness, and who should live, who die, and so on. It also can randomly fire cosmic rays at cells, or make them crossover with others. It also implements the smallest time segment, a generation. This abstract class has reasonable default behaviours in place for these things, but they can be overridden by extending if desired. As the fitness function is domain-specific, it must be implemented by extending classes.

```
import java.util.*;
```

```
public abstract class Environment {
```

- *ArrayList cells*
Stores the collection of **Cell** objects. The list should be kept sorted with the highest fitness at the lowest index.
- *double cosmicRayFreq*
The probability that a particular cell will get hit by a cosmic ray per generation. When a cell is hit, its *mutate()* method is called.
- *int crossoverRate*
This specifies the number of new cells that will be created by crossover.
- *boolean sorted*
This is true when the lists of cells is sorted by fitness. Anything that alters the list without ensuring the order should set this to false and a sort will be performed at the next generation.
- *int ID_LENGTH=10*
This sets the length of the ID strings given to the **Genomes**.
- *char[] lastID*
The ID number is stored here.

```
ArrayList cells;  
double cosmicRayFreq;  
int crossoverRate;  
boolean sorted=false;  
int ID_LENGTH=10;  
char[] lastID;
```

B.1.1 Constructor

The default constructor creates an empty set of cells, and turns off cosmic rays. This is suitable for inserting cells into from somewhere else.

```
public Environment() {  
    cells = new ArrayList();  
    cosmicRayFreq=0.0;  
}
```

This constructor starts the environment off with a number of cells. This just calls an abstract method, which must be overridden to actually do it.

```
public Environment(int s) {  
    initCells(s);  
}
```

This is the constructor that will typically be used when setting up a new system. It sets the size, the cosmic ray rate, and the crossover rate.

```

public Environment(int s, double r, int c) {
    this(s);
    cosmicRayFreq = r;
    crossoverRate = c;
}

```

B.1.2 initCells (abstract)

This method needs to create the **ArrayList** to put the cells in, and put them there.

```

protected abstract void initCells(int size);

```

B.1.3 generation

This function does the work required for each generation:

1. Check to see if the list is sorted (and sort it if need be)
2. Perform crossover amongst the cells
3. Throw in some cosmic mutation

In most part, other methods are called to carry these out, this allows the behaviour to be easily modified by extending classes. To add extra functionality (for example storing cells to disk, recording statistics, etc.) this method can be wrapped by an extending one. This method is called by the controlling program.

```

public void generation() {
    if (!sorted) {
        sortCellList();
    }
    crossover();
    int s = cells.size();
    for (int i=0; i<s; i++) {
        if (random() <= cosmicRayFreq) {
            ((Cell)(cells.get(i))).mutate();
        }
    }
    clearConvergence();
}

```

B.1.4 sortCellList

This method is an internal one that sorts the list according to the fitness function. It does this by creating a new list, and stepping through and inserting each object into the correct place in the new list. The list should be ordered from highest to lowest.

```

protected void sortCellList() {
    ArrayList tempCells = new ArrayList(cells.size());
    int top, bot, cent;
    int s = cells.size();
    bot = 0;
    top = 0;
    cent = 0;
    for (int i=0; i<s; i++) {
        top = i;
        Cell tempCell = (Cell)cells.remove(0);
    }
}

```

```

        double f = fitness(tempCell);
        if ((i==0) || (fitness((Cell)tempCells.get(i-1))>=f))
            tempCells.add(tempCell);
        else {
            if (fitness((Cell)tempCells.get(0))<=f)
                tempCells.add(0,tempCell);
            else {
                int insPoint=0;
                for (int j=0; j<i; j++) {
                    if (fitness((Cell)tempCells.get(j)) < f) {
                        insPoint = j;
                        j=i;
                    }
                }
                tempCells.add(insPoint, tempCell);
            }
        }
    }
    cells = tempCells;
}

```

B.1.5 fitness (Abstract)

This is an abstract method that takes a **Cell** and returns a real number representing the fitness value of the cell. The higher the number, the more fit the cell is. This is the main domain-specific part of the program.

```

public abstract double fitness(Cell c);

```

B.1.6 crossover

This method performs the selection and crossover procedure. As implemented here, it simply selects twice the number of cells from the top of the list (i.e the high fitness ones) as it wants to produce (from *crossoverRate*) and randomly does crossover on pairs. These new ones are then placed at the bottom of the list, replacing the low fitness ones.

This method needs to be overridden or wrapped if more complex crossover selection or operations are wanted.

```

public void crossover() {
    ArrayList parentsList = new ArrayList(crossoverRate*2);
    ArrayList childrenList = new ArrayList(crossoverRate);
    int s = crossoverRate*2;
    for (int i=0; i<s;i++)
        parentsList.add(cells.get(i));
    for (int i=0; i<crossoverRate; i++) {
        int c = (int)random(s);
        childrenList.add(((Cell)parentsList.get(c)).copy(getNextID()));
        ((Cell)childrenList.get(i)).crossover(((Cell)parentsList.get((int)random(s))));
    }
}

```

Now that we have built up a list of new organisms, we need to insert them into the environment. For this implementation, they go at the end of the list.

```

int t=cells.size()-crossoverRate;
for (int i=0; i<crossoverRate; i++) {

```

```

        cells.set(i+t, childrenList.get(i));
    }
    sorted = false;
}

```

We need a couple of random number generators to support this.

B.1.7 random (zero arguments)

This returns a double in the range $0 \leq x < 1$

```

public double random() {
    return Math.random();
}

```

B.1.8 random (one argument)

This returns a double in the range $0 \leq x < r$. Note that this is simply the *random()* (Section B.1.7) multiplied, so there is no precision gained over it.

```

public double random(double r) {
    return random()*r;
}

```

B.1.9 getNextID

This fetches the next unique ID string for an organism, returns it, and increases the ID counter. A more efficient method of doing this should be found. This method ignores wraparound, but if *ID_LENGTH* is set sufficiently high (say, 10) then this shouldn't be a problem.

```

public String getNextID() {
    if (lastID!=null) {
        lastID[ID_LENGTH-1]++;
        boolean carry=false;
        for (int i=ID_LENGTH-1; i>=0; i--) {
            if (carry)
                lastID[i]++;
            if (lastID[i]<='Z')
                carry = false;
            else {
                lastID[i]='A';
                carry = true;
            }
        }
    } else {
        lastID = new char[ID_LENGTH];
        for (int i=0; i<ID_LENGTH; i++)
            lastID[i] = 'A';
    }
    return new String(lastID);
}

```

B.1.10 getFitnesses

This method returns an array that contains all the fitness values of the system as it is at the moment. Because the sorting happens at the start of the generation, if sorted values are wanted, then specify 'true'. If it doesn't matter, specify false.

```
public double[] getFitnesses(boolean sort) {
    if (sort && !sorted) {
        sortCellList();
    }
    double [] foo = new double[cells.size()];
    for (int i=0; i<foo.length; i++) {
        foo[i] = fitness((Cell)cells.get(i));
    }
    return foo;
}
```

B.1.11 getCellConvergence (abstract)

Convergence is a measure of how alike one cell is to all the others. The simplest way to calculate is to build a frequency table of what value is found in what location. Then, the **Genotype** of the cell in question is compared to that table, and for each location in the genotype, the proportion of the population that share it is calculated. The mean of these is then taken, and this is the convergence of the cell.

There are two versions, for convenience. One gets the convergence of an arbitrary cell, the other gets the other gets the convergence of a cell in the environment, specified by a number, and an option to ensure the cell list is sorted first.

```
public abstract double getCellConvergence(Cell c);

public double getCellConvergence(int n, boolean sort) {
    if (!sorted && sort) {
        sortCellList();
    }
    return getCellConvergence((Cell)cells.get(n));
}
```

B.1.12 getPopnConvergence

This returns the average convergence level across the whole population. This is done by calculating the convergence for every organism and taking the mean. As it is here, this will work, but it may be a good idea to override it with a more domain-specific version which will likely be able to squeeze out a bit more efficiency.

```
public double getPopnConvergence() {
    int s = cells.size();
    double c = 0.0;
    for (int i=0; i<s; i++) {
        c += getCellConvergence((Cell)cells.get(i));
    }
    return s==0 ? 0 : c/s;
}
```


B.1.13 clearConvergence (abstract)

Because it would be computationally expensive to calculate the lookup table for every cell we want to test, it is good to cache it, and reset it every generation. This method is called each generation to do the reset.

```
public abstract void clearConvergence();
```

For good measure, we have a few accessors and modifiers. These can be used to alter the behaviour of the system as it is running.

B.1.14 getCosmicRayFreq

Returns the frequency of cosmic ray hits.

```
public double getCosmicRayFreq() {  
    return cosmicRayFreq;  
}
```

B.1.15 setCosmicRayFreq

Sets the cosmic ray level of the universe.

```
public void setCosmicRayFreq(double c) {  
    cosmicRayFreq = c;  
}
```

B.1.16 getCrossoverRate

Gets the crossover rate.

```
public int getCrossoverRate() {  
    return crossoverRate;  
}
```

B.1.17 setCrossoverRate

Sets the crossover rate.

```
public void setCrossoverRate(int c) {  
    crossoverRate = c;  
}
```

B.1.18 getCell

This retrieves a cell from the environment. If the method is given a “true” argument, it ensures that the list is sorted first.

```
public Cell getCell(int c, boolean sort) {  
    if (sort && !sorted) {  
        sortCellList();  
    }  
    return (Cell)cells.get(c);  
}
```

B.1.19 setCell

This allows a cell to be inserted into the environment, replacing what was there before. It has three arguments: index, **Cell** to insert, and a boolean stating whether the list should be sorted first.

```
public void setCell(int i, Cell c, boolean sort) {  
    if (sort && !sorted) {  
        sortCellList();  
    }  
    cells.set(i,c);  
}
```

B.2 Cell class (abstract)

This class loosely represents a cell, in that it holds a genome. Operations such as the storing of non-genetic information (e.g. parent information) and storage of the fitness function (so that it doesn't need to be calculated again if nothing has changed) are done here. Mutation and crossover are done in anything extending this.

```
import java.util.*;
```

```
public abstract class Cell {
```

- *Genome DNA*
Stores the “genetic material” of the cell.
- *String ID*
Stores the identifier for this cell, used to identify it uniquely.
- *ArrayList parents*
Stores a list provided by the environment that is used to build a “family history” of cells.
- *double fitness*
The fitness function of the cell, as provided by the environment. This is used to avoid unnecessary recalculation.
- *boolean dirtyBit*
True if the fitness is invalid (e.g. the genome has been altered)

```
    Genome DNA;  
    String ID;  
    ArrayList parents;  
    double fitness;  
    boolean dirtyBit = true;
```

B.2.1 Constructor

The constructor for this can accept a **Genome** object, and use that. There is also a constructor, to be overridden, that accepts a size. Both of these accept a **String** that represents the unique identifier, provided by the environment, and an **ArrayList** that is a list of parents.

```
    protected Cell() {}  
  
    protected Cell(String i, ArrayList p) {  
        ID = i;  
        parents = p;  
    }  
  
    public Cell(String i, ArrayList p, Genome g) {  
        this(i,p);  
        DNA = g;  
    }  
  
    protected Cell(String i, ArrayList p, double f,  
                    boolean d, Genome g) {  
        this(i,p,g);  
        fitness = f;  
        dirtyBit = d;  
    }
```

B.2.2 mutate (abstract)

This method, which must be implemented by the extending class, is called when this cell has been chosen to be mutated. As what it does is dependant on the structure of the genome data itself, it has to be left to a class which knows about that. Should make *dirtyBit* true.

```
public abstract void mutate();
```

B.2.3 crossover (abstract)

This method accepts an argument of another **Cell** object, and alters itself to be the result of crossover between itself and the **Cell** provided. Should make *dirtyBit* true.

```
public abstract void crossover(Cell c);
```

B.2.4 getID

This method returns the ID of this cell. The ID is whatever it was told by the constructor, or was set by the *setID()* method (Section B.2.5).

```
public String getID() {  
    return ID;  
}
```

B.2.5 setID

This sets the ID, replacing whatever was set by the constructor.

```
public void setID(String i) {  
    ID = i;  
}
```

B.2.6 getParents

This gets the list of parents of the cell.

```
public ArrayList getParents() {  
    return parents;  
}
```

B.2.7 setParents

This replaces the existing list of parents with the one supplied.

```
public void setParents(ArrayList a) {  
    parents = a;  
}
```

B.2.8 getGenome

This retrieves the genome of the cell.

```
public Genome getGenome() {  
    return DNA;  
}
```

B.2.9 setFitness

This sets the fitness value of the cell.

```
public void setFitness(double f) {
    fitness = f;
    dirtyBit = false;
}
```

B.2.10 getFitness

Retrieves the fitness value.

```
public double getFitness() {
    return fitness;
}
```

B.2.11 dirty

Returns the status of *dirtyBit*, indicating whether or not the fitness value is to be trusted.

```
public boolean dirty() {
    return dirtyBit;
}
```

B.2.12 makeDirty

Makes a cell “dirty”, by setting *dirtyBit*.

```
public void makeDirty() {
    dirtyBit = true;
}
```

B.2.13 makeClean

Makes a cell “clean”, by resetting the *dirtyBit*.

```
public void makeClean() {
    dirtyBit = false;
}
```

B.2.14 copy (abstract)

This duplicates a cell, creating new instances of everything contained in it. It gives it the ID specified. Note that things like the parent list are kept intact, and may need to be altered depending on the application. It must be overridden, and look something like: `return new Cell(newID, parents.clone(), fitness, dirtyBit, DNA.copy());` except with `Cell` replaced by whatever the subclass is called.

```
public abstract Cell copy(String newID);
}
```

B.3 Genome class (abstract)

This class is quite simple, it is the abstract level of interface to a gene. As much of this is implementation specific, there is very little required here, and some casting will have to be done by the program that uses it to ensure that the correct functions are available.

```
public abstract class Genome {
```

The data fields global to all **Gene** extenders go here.

- *int size*
Defines the number of elements in the genome. What this means is really up to the extending class.

```
int size;
```

B.3.1 Constructor

The ever-present default constructor.

```
public Genome(){}
```

The constructor accepts a size, and stores it. It is very likely that this will be overridden in order to actually set up the contents of the genotype.

```
public Genome(int s) {  
    size = s;  
}
```

B.3.2 randomise (abstract)

The method *randomise()* should make the sequence random.

```
public abstract void randomise();
```

B.3.3 copy (abstract)

This does a deep copy of this object.

```
public abstract Genome copy();  
}
```

Appendix C

Source Code For Application

C.1 Test Program 2

This code is an extension of the first test program (which isn't included in this report, as its functionality is a subset of this one). The problem domain is the same, however this version allows multiple populations to be created.

The multiple population system is handled with an extension to the syntax. The environment variable 'defpop' specifies the default population. Any command prefixed with **n:** will operation on population **n**, for example: **3:create** will perform the **create** operation on population 3. The variable 'numpops' defines the upper bound of this number.

This also supports ranges (e.g. **2-4:create**). A wildcard is also allowed, e.g. ***:create**.

Variables that have a meaning local to the population are affected by the prefix (for example 'crossrate'), others are independent of it ('fitcap').

```
import java.io.*;
import java.util.*;
import java.text.*;

public class Test2 {
    int varDispFreq;
    double varConvCap;
    int varConvCheck;
    double varFitCap;
    int varFitCheck;
    int varNumPop = 1;
    int varDefPop;
    int varMigRate;
    int varMigFreq;
    int varMigPolicy;
    int numGens;
    ArrayList environments = new ArrayList();
    Hashtable cellStorage;
    static final String startFile = "startup.ga";
```

C.1.1 main

This is the entry point into the program. Initially, it creates the instances of the programs, and then calls the command line interface.

```
public static void main(String[] args) {
```

First we load **startup.ga** if it exists, then go to the command line. The loading must be done in a slightly roundabout way, because *loadScript* expects a list of options as its input.

```
    Test2 foo = new Test2();
    ArrayList f = new ArrayList();
    f.add(startFile);
    foo.loadScript(f, System.out);
    foo.doCommandLine();
}
```

C.1.2 Constructor

This constructor is called when the basic object is created, and is used to set up stuff that should be applied at startup.


```

public Test2() {
    varNumPop = 1;
    resizePopns();
    cellStorage = new Hashtable();
}

```

C.1.3 doCommandLine

This is a basic command line interface to do things like set the GA system parameters, start it running and extract statistics. It will likely be replaced or supplimented by a more flexible system eventually.

```

public void doCommandLine() {
    boolean quit = false;
    String cmdLine="";
    String lastCmd="";
    BufferedReader input =
        new BufferedReader(new InputStreamReader(System.in));
    while (!quit) {
        System.out.print("GA> ");
        try {
            cmdLine = input.readLine();
        } catch (IOException e) {
            System.out.println("Input error: "+e.getLocalizedMessage());
        }
        // A (mis)feature in Java means that a readLine() returns
        // null if ctrl-D is pressed.
        if (cmdLine == null) {
            System.out.println("exit");
            cmdLine = "exit";
        }
    }
}

```

For quick repeating of commands.

```

    if (cmdLine.equals("!!")) {
        cmdLine = lastCmd;
    }

```

Parse the command line and call the relevant methods.

```

    int res = parseCmdLine(cmdLine, System.out);
    if (res == -1) {
        quit = true;
    } else if (res == 1) {
        System.out.println("Error: No such command: "+cmdLine);
    }
    lastCmd = cmdLine;
}
}

```

C.1.4 parseCmdLine

This takes a string and parses it as though it is a command line. This allows things like recursive loading of scripts to be implemented.

Return values are the same as for *dispatch* (Section C.1.6), with -1 meaning to end the program, 0 meaning success, and 1 meaning there was an error.

```
public int parseCmdLine(String cmdLine, PrintStream defStream) {
```

To parse the line we need a tokenizer that knows about ‘;’.

```
StringTokenizer st = new StringTokenizer(cmdLine, " \t\n\r\f>");
```

If there are no tokens, then just ignore.

```
if (st.countTokens() != 0) {
```

Now wander through and see if we need redirection.

```
PrintStream output = defStream;
int n;
if ((n = cmdLine.indexOf('>')) != -1) {
```

If this is followed by another one, we need to append, not overwrite.

```
boolean append = false;
if (cmdLine.charAt(n+1) == '>') {
    n++;
    append = true;
}
try {
    output =
        new PrintStream(new FileOutputStream(cmdLine.
                                            substring(n+1).
                                            trim(),append));
} catch (IOException e) {
```

If there was a problem, we say, and then just carry on as though nothing had happened (output goes to System.out)

```
defStream.println("IO Error: "+e.getLocalizedMessage());
}
```

We also need to take the left hand side and make it the command.

```
cmdLine = cmdLine.substring(0,n);
}
ArrayList envs = new ArrayList();
```

Now look at that command and see if it has any sort of range specifier. If it does, build a list of the populations to cover.

```
if ((n = cmdLine.indexOf(':')) != -1) {
    try {
        String range = cmdLine.substring(0,n).trim();
        cmdLine = cmdLine.substring(n+1);
        if (range.length() == 0) {
            defStream.println("' :' without numbers doesn't make "+
                            "sense.");
            return 0;
        }
        if (range.equals("*")) {
            // All values
            for (int i=0; i<environments.size(); i++) {
                envs.add(environments.get(i));
            }
        }
    }
}
```

```

    }
} else {
    if ((n = range.indexOf('-')) != -1) {
        // A range
        int num1 = Integer.parseInt(range.substring(0,n));
        int num2 = Integer.parseInt(range.substring(n+1));
        if ((num1<0) ||
            (num2>=environments.size()) ||
            (num1>=num2)) {
            defStream.println("Invalid environment "+
                             "range specified.");
        }
        for (int i=num1; i<=num2; i++) {
            envs.add(environments.get(i));
        }
    } else {
        // A single value
        int envNum = Integer.parseInt(range);
        if ((envNum>=environments.size()) || (envNum<0)) {
            defStream.println("Invalid environment "+
                             "specified.");
            return 0;
        }
        envs.add(environments.get(Integer.
                                parseInt(range)));
    }
}
} catch (NumberFormatException e) {
    defStream.println("Environment numbering error: "+
                     e.getLocalizedMessage());
    return 0;
}
} else
    envs.add(environments.get(varDefPop));
if (envs.size() == 0) {
    defStream.println("No valid environments specified.");
    return 0;
}
}

```

Now we need to take the first token, which is the actual command.

```

st = new StringTokenizer(cmdLine, " \t\n\r\f>");

String cmd = st.nextToken();

```

Now figure out its options, put into an ArrayList containing strings.

```

ArrayList options = new ArrayList();

```

Add a dummy entry that will have the environment placed into it.

```

options.add(null);
while (st.hasMoreTokens())
    options.add(st.nextToken());
if (cmd.equals("@load")) {
    Exception e = loadScript(options,defStream);
}

```

```

        if (e != null) {
            defStream.println("@load: Error: "+
                               e.getLocalizedMessage());
        }
    } else {
        int res = 0;
        for (int i=0;i<envs.size() && 0 == res; i++) {
            options.set(0,envs.get(i));
            res = dispatch(cmd,output,options);
        }
        return res;
    }
}

```

There was no input, or it was internal and didn't fail.

```

        return 0;
    }
}

```

C.1.5 loadScript

This loads a script file by calling *parseCmdLine* with each line in the file. Returns an exception if there is an error, otherwise null.

```

public Exception loadScript(ArrayList opts, PrintStream defStream) {
    if (opts.size() == 1) {
        try {
            BufferedReader input =
                new BufferedReader(new FileReader((String)opts.get(0)));
            String s;
            while ((s=input.readLine()) != null) {
                if ((s.length() != 0) && (s.charAt(0) != '#')) {
                    int res=parseCmdLine(s,defStream);
                    if (res == -1) {
                        System.exit(0);
                    } else if (res == 1) {
                        defStream.println("Error: invalid command: "+s);
                    }
                }
            }
        } catch (IOException e) {
            return e;
        }
    } else {
        defStream.println("Error: @load requires a filename to load.");
    }
    return null;
}

```

C.1.6 dispatch

This handles the execution of commands. It is to allow an easy way to handle things like scripting from a file as well as interactive commands. Returns -1 to finish the program, 0 for success, 1 for invalid command.

```

public int dispatch(String cmd, PrintStream output, ArrayList options) {

    if (cmd.equals("set"))
        cmdSet(output,options);
    else if (cmd.equals("create"))
        cmdCreate(output,options);
    else if (cmd.equals("destroy"))
        cmdDestroy(output,options);
    else if (cmd.equals("run"))
        cmdRun(output,options);
    else if (cmd.equals("display"))
        cmdDisplay(output,options);
    else if (cmd.equals("stats"))
        cmdStats(output,options);
    else if (cmd.equals("extract"))
        cmdExtract(output,options);
    else if (cmd.equals("replace"))
        cmdReplace(output,options);
    else if (cmd.equals("reset"))
        cmdReset(output,options);
    else if (cmd.equals("quit") ||
            cmd.equals("q") ||
            cmd.equals("exit"))
        return -1;
    else
        return 1;
    return 0;
}

```

C.1.7 cmdSet

This implements the **set** command, which sets parameters that affect the functioning of the system. This probably should be done in a better way, perhaps a hashtable to allow arbitrary variables to be created and used. Maybe another time. Variables are:

- *popn*
The number of organisms stored. Cannot be changed in an active system.
- *cosmic*
Cosmic ray frequency, as defined in the **Environment** class (Section B.1)
- *crossrate*
The crossover rate.

It is important to note that the **Environment** class does not query us for the values, so any relevant changes must be pushed through to it.

```

public void cmdSet(PrintStream o, ArrayList opts) {

```

Initially, we'll copy the environment wrapper into a more convenient variable.

```

    EnvWrapperT2 env = (EnvWrapperT2)opts.get(0);

```

If there were no parameters, we list everything we know about.

```

    if (opts.size() == 1) {
        o.println("set popn "+env.getVarPopn());
        o.println("set cosmic "+env.getVarCosmic());
    }

```

```

o.println("set crossrate "+env.getVarCrossrate());
o.println("set dispfreq "+varDispfreq);
o.println("set convcap "+varConvCap);
o.println("set convcheck "+varConvCheck);
o.println("set fitcap "+varFitCap);
o.println("set fitcheck "+varFitCheck);
o.println("set defpop "+varDefPop);
o.println("set numpop "+varNumPop);
o.println("set migrate "+varMigRate);
o.println("set migfreq "+varMigFreq);
o.println("set migpolicy "+varMigPolicy);
} else if (opts.size() == 2) {

```

If there was only one parameter, we list it.

```

if ("popn".equals((String)opts.get(1))) {
    o.println("set popn "+env.getVarPopn());
} else if ("cosmic".equals((String)opts.get(1))) {
    o.println("set cosmic "+env.getVarCosmic());
} else if ("crossrate".equals((String)opts.get(1))) {
    o.println("set crossrate "+env.getVarCrossrate());
} else if ("dispfreq".equals((String)opts.get(1))) {
    o.println("set dispfreq "+varDispfreq);
} else if ("convcap".equals((String)opts.get(1))) {
    o.println("set convcap "+varConvCap);
} else if ("convcheck".equals((String)opts.get(1))) {
    o.println("set convcheck "+varConvCheck);
} else if ("fitcap".equals((String)opts.get(1))) {
    o.println("set fitcap "+varFitCap);
} else if ("fitcheck".equals((String)opts.get(1))) {
    o.println("set fitcheck "+varFitCheck);
} else if ("defpop".equals((String)opts.get(1))) {
    o.println("set defpop "+varDefPop);
} else if ("numpop".equals((String)opts.get(1))) {
    o.println("set numpop "+varNumPop);
} else if ("migrate".equals((String)opts.get(1))) {
    o.println("set migrate "+varMigRate);
} else if ("migfreq".equals((String)opts.get(1))) {
    o.println("set migfreq "+varMigFreq);
} else if ("migpolicy".equals((String)opts.get(1))) {
    o.println("set migpolicy "+varMigPolicy);
} else o.println("Error: no such variable.");
} else {

```

However, as there is more than one parameter, we set the variable. We may also need to push this value to the **EnvWrapperT2** class.

```

try {
    if ("popn".equals((String)opts.get(1))) {
        if (env.getEnv() == null) {
            env.setVarPopn(Integer.parseInt((String)opts.get(2)));
        } else {
            o.println("Error: value cannot be changed on an "+
                "existing system");
        }
    }
    } else if ("cosmic".equals((String)opts.get(1))) {

```

```

        env.setVarCosmic(Double.parseDouble((String)opts.get(2)));
        if (env.getEnv() != null) {
            env.getEnv().setCosmicRayFreq(env.getVarCosmic());
        }
    } else if ("crossrate".equals((String)opts.get(1))) {
        env.setVarCrossrate(Integer.parseInt((String)opts.get(2)));

        if (env.getEnv() != null) {
            env.getEnv().setCrossoverRate(env.getVarCrossrate());
        }
    } else if ("dispfreq".equals((String)opts.get(1))) {
        varDispFreq = Integer.parseInt((String)opts.get(2));
    } else if ("convcap".equals((String)opts.get(1))) {
        varConvCap = Double.parseDouble((String)opts.get(2));
    } else if ("convcheck".equals((String)opts.get(1))) {
        varConvCheck = Integer.parseInt((String)opts.get(2));
    } else if ("fitcap".equals((String)opts.get(1))) {
        varFitCap = Double.parseDouble((String)opts.get(2));
    } else if ("fitcheck".equals((String)opts.get(1))) {
        varFitCheck = Integer.parseInt((String)opts.get(2));
    } else if ("defpop".equals((String)opts.get(1))) {
        varDefPop = Integer.parseInt((String)opts.get(2));
    } else if ("numpop".equals((String)opts.get(1))) {
        int v = Integer.parseInt((String)opts.get(2));
        if (v<0) {
            o.println("Error: numpop can't be set <0.");
        } else {
            varNumPop = Integer.parseInt((String)opts.get(2));
            resizePopns();
        }
    } else if ("migrate".equals((String)opts.get(1))) {
        varMigRate = Integer.parseInt((String)opts.get(2));
    } else if ("migfreq".equals((String)opts.get(1))) {
        varMigFreq = Integer.parseInt((String)opts.get(2));
    } else if ("migpolicy".equals((String)opts.get(1))) {
        varMigPolicy = Integer.parseInt((String)opts.get(2));
    } else o.println("Error: no such variable.");
} catch (NumberFormatException e) {
    o.println("Error: invalid value: "+e.getLocalizedMessage());
}
}
}

```

C.1.8 resizePopns

This ensures that the size of the *environments* **ArrayList** matches the value of *varNumPop*.

```

public void resizePopns() {
    if (varNumPop < environments.size()) {
        for (int i=environments.size(); i>=varNumPop; i--) {
            environments.remove(i);
        }
    } else if (varNumPop > environments.size()) {
        for (int i=environments.size(); i<varNumPop; i++)
            environments.add(new EnvWrapperT2());
    }
}

```

```

    }
    environments.trimToSize();
}

```

C.1.9 cmdCreate

This creates an instance of the **EnvWrapperT2** class with an **EnvironmentT2** inside it, and starts it with the appropriate values, according to the variables that are set. This starts the population with a set of randomised organisms.

```

public void cmdCreate(PrintStream o, ArrayList opts) {
    EnvWrapperT2 env = (EnvWrapperT2)opts.get(0);
    env.create();
}

```

C.1.10 cmdDestroy

This removes the environment from memory, allowing a new one to be created.

```

public void cmdDestroy(PrintStream o, ArrayList opts) {
    ((EnvWrapperT2)opts.get(0)).setEnv(null);
}

```

C.1.11 cmdRun

This runs the system for a specified number of generations. Statistics are displayed every so often, determined by the *varDispfreq* variable.

This method pretty much ignores the list of environments, and it operates on all environments that are set up. A way to activate and deactivate environments will be added if necessary.

NEVER EVER call the run command with a range of environments, it will be executed several times, and this isn't a good thing.

```

public void cmdRun(PrintStream o, ArrayList opts) {
    boolean done = false;

```

First check to ensure that at least one environment has been created in order to run. At the same time, we'll plug them into an ArrayList for later.

```

    ArrayList envsToRun = new ArrayList();
    Hashtable envNum = new Hashtable();
    EnvWrapperT2 e;
    for (int i=0; i<environments.size(); i++) {
        if ((e=(EnvWrapperT2)environments.get(i)).getEnv() != null) {
            envsToRun.add(e);
            envNum.put(((EnvWrapperT2)e).getEnv(),new Integer(i));
        }
    }
    if (envsToRun.size() == 0) {
        o.println("Error: A system needs to be created before "+
            "it can be run.");
        return;
    }
}

```

Have we been supplied with a number of generations to go for?


```

if (opts.size() != 2) {
    o.println("Error: A number of generations to run must be given.");
    return;
}

```

Work out how many generations we want to run for. (It's not really too hard)

```

int wantedGens;
try {
    wantedGens = Integer.parseInt((String)opts.get(1));
} catch (NumberFormatException ex) {
    o.println("Error: invalid number of generations: "+
        ex.getLocalizedMessage());
    return;
}
for (int i=0; (i<wantedGens && !done); i++) {
    Iterator envIt = envsToRun.iterator();
    if ((varMigRate != 0) &&
        (varMigFreq != 0) &&
        (envsToRun.size()>1)) {
        if (numGens % varMigFreq == 0) {

```

First we need to split off depending on migration policy. If the variable *varMigPolicy* is 0, then we do a simple form of migration, where the best from each population are “cloned”, and sent off to all the other ones.

If *varMigPolicy* is 1, then we emulate a biological system, where the organisms aren't actually cloned, but are moved. This actually alters quite a bit, in the original method, if the migration rate was two and there were three populations, then each population would get six new organisms. In this variation it only gets two. They are spread out as much as is possible, also. The actual code for this starts a few blocks down.

```

int numPops=envsToRun.size();
if (varMigPolicy == 0) {

```

Here we need to do basic migration. Basically we pull the best few from each population, put them into an array, and then go through and insert each one into all the other **Environments**.

```

Cell[][] migrants = new Cell[numPops][varMigRate];

```

First up we go through the population and take the best *varmigRate* populations, and insert them into the array.

```

for (int en=0;en<numPops;en++) {
    EnvWrapperT2 envw =
        (EnvWrapperT2)envsToRun.get(en);
    for (int f=0;f<varMigRate;f++)
        migrants[en][f] =
            envw.getEnv().getCell(f,true).copy("");
}

```

Now that is done, we go through each array and insert everything from it into the bottom end of each environment (except the one it came from).

The loops get a little messy here. The first one steps over each population telling us what we are inserting to, the second one steps over each population in the array telling us part of what we are taking out, and the third steps over the amount from each population.

```

        for (int en=0;en<numPops;en++) {
            EnvWrapperT2 envw =
                (EnvWrapperT2)envsToRun.get(en);
            int insertP = envw.getVarPopn()-1;
            for (int f=0;f<numPops;f++)
                if (en!=f)
                    for (int g=0;g<varMigRate;g++)
                        envw.getEnv().
                            setCell(insertP--,
                                migrants[f][g].copy(""),true);
        }
    } else if (varMigPolicy == 1) {

```

Here we do the more biologically-inspired method. In this, all the best ones are collected up, and then the contents of that array is spread out amongst all the populations (except the one it originated from).

```

        Cell[] migrants = new Cell[varMigRate*numPops];
        int arrayPtr=0;
        for (int p=0; p<numPops; p++) {
            EnvWrapperT2 envw =
                (EnvWrapperT2)envsToRun.get(p);
            for (int org=0;org<varMigRate;org++) {
                migrants[arrayPtr++] =
                    envw.getEnv().getCell(org,true).copy("");
            }
        }
    }

```

Now we go and place each organism in this array into another population, taking care that it isn't inserted into the one that it came from. Note that this inserts from the top which effectively removes the organism that left.

```

        int destPop = 0;
        arrayPtr = 0;
        int[] popInsPt = new int[numPops];
        for (int j=0;j<numPops;j++) {popInsPt[j]=0;}
        while (arrayPtr<migrants.length) {
            if ((arrayPtr<destPop*varMigRate) ||
                (arrayPtr>=(destPop+1)*varMigRate)) {
                EnvWrapperT2 envw =
                    (EnvWrapperT2)envsToRun.get(destPop);
                envw.getEnv().
                    setCell(popInsPt[destPop]++,
                        migrants[arrayPtr++],false);
            }
            destPop=(destPop+1)%numPops;
        }
    }
}

```

Here each environment is stepped through, and its *generation()* method is called. Other checks, such as checking the various limits and whether to display are done after that.

```

while (envIt.hasNext()) {
    Environment env = ((EnvWrapperT2)envIt.next()).getEnv();

```

```

env.generation();
if ((varConvCheck != 0) &&
    (numGens % varConvCheck == 0) &&
    (env.getPopnConvergence() >= varConvCap))
    done = true;
double avg = 0.0;
double max=-Double.MAX_VALUE;
if (varFitCheck != 0)
    if (numGens % varFitCheck == 0) {
        double [] stats = env.getFitnesses(true);
        double sum = 0.0;
        for (int j=0; j<stats.length; j++) {
            sum += stats[j];
            if (stats[j]>max)
                max=stats[j];
        }
        avg = sum / stats.length;
        if (avg >= varFitCap)
            done = true;
    }
if (varDispfreq != 0)
    if ((numGens % varDispfreq == 0) || done) {
        if (avg == 0.0) {
            double [] stats = env.getFitnesses(true);
            double sum = 0.0;
            for (int j=0; j<stats.length; j++) {
                sum += stats[j];
                if (stats[j]>max)
                    max=stats[j];
            }
            avg = sum / stats.length;
        }
        int number=((Integer)envNum.get(env)).intValue();
        o.println(number+"\t"+
            numGens+"\t"+
            avg+"\t"+
            max+"\t"+
            toDP(5,env.getPopnConvergence()));
    }
}
numGens++;
}
}

```

C.1.12 cmdExtract

This method allows a representation of an organism to be displayed.

```

public void cmdDisplay(PrintStream o, ArrayList opts) {
    if (opts.size() == 1) {
        o.println("Error: a list of organisms must be specified.");
        return;
    }
    EnvWrapperT2 wrapper = (EnvWrapperT2)opts.get(0);
    EnvironmentT2 env = wrapper.getEnv();
}

```

```

    if (env == null) {
        o.println("Error: a system needs to be created first.");
        return;
    }
    for (int i=1; i<opts.size(); i++) {
        int cellNum;
        try {
            cellNum = Integer.parseInt((String)opts.get(i));
        } catch (NumberFormatException e) {
            o.println("Error: not a number: "+e.getLocalizedMessage());
            continue;
        }
        if (cellNum > wrapper.getVarPopn()) {
            o.println("Error: there is no such cell "+cellNum);
            continue;
        }
        CellT2 cell = (CellT2)env.getCell(cellNum,false);
        GenomeT2 gen = (GenomeT2)cell.getGenome();
        byte[] geneString = gen.getGene();
        o.println("ID: "+cell.getID());
        o.println("Convergence: "+env.getCellConvergence(cell));
        o.println("Parents:");
        ArrayList p = cell.getParents();
        for (int j=0; j<p.size(); j++)
            o.println(" > "+(String)p.get(j));
        for (int j=0; j<geneString.length; j++)
            o.print(byteToHex(geneString[j])+" ");
        o.println();
    }
}

```

C.1.13 cmdStats

This displays a detailed bunch of statistics regarding the state of of the system and the cells in it.

```

public void cmdStats(PrintStream o, ArrayList opts) {
    Environment env = ((EnvWrapperT2)opts.get(0)).getEnv();
    if (env == null) {
        o.println("Error: the environment must be created first.");
        return;
    }
    o.print(numGens+" {");
    double[] stats = env.getFitnesses(true);
    double sum = 0.0;
    for (int i=0; i<stats.length; i++) {
        o.print(" "+stats[i]+", "+toDP(5,env.getCellConvergence(i,false)));
        sum += stats[i];
    }
    o.println(" } "+sum/stats.length+" "+
        toDP(5,env.getPopnConvergence()));
}

```

C.1.14 cmdExtract

This command allows individual **Cells** to be copied from the current environment, and placed in a global pool. The first option given is the index of the genome to extract, starting from zero. For conveniences sake, negative indexes work from the other end of the **Environment**, so that 0 is the best, and -1 is the worst. The identifier used to store them is any string, and a hashtable is the storage method.

```
public void cmdExtract(PrintStream o, ArrayList opts) {
    Environment env = ((EnvWrapperT2)opts.get(0)).getEnv();
    if (env == null) {
        o.println("Error: the environment must be created first.");
        return;
    }
    if (opts.size() != 3) {
        o.println("Error: insufficient parameters given.");
        return;
    }
}
```

First work out what cell we want, and make sure it is valid.

```
int numCells = ((EnvWrapperT2)opts.get(0)).getVarPopn();
int wantedCell;
try {
    wantedCell = Integer.parseInt((String)opts.get(1));
} catch (NumberFormatException e) {
    o.println("Error: invalid cell number: "+e.getMessage());
    return;
}
if (wantedCell < 0)
    wantedCell = numCells+wantedCell;
if (wantedCell >= numCells) {
    o.println("Error: cell number out of range: "+wantedCell);
    return;
}
```

Now get the key that it is going to be stored with.

```
String key = (String)opts.get(2);
```

Now do it all...(note that we're not caring about family trees here, so don't bother with IDs.)

```
Cell c = (env.getCell(wantedCell,true)).copy("");
cellStorage.put(key,c);
}
```

C.1.15 cmdReplace

This is the counterpart to *cmdExtract*. It takes a cell from the storage **Hashtable** and puts it into the **Environment**. All the arguments are as *cmdExtract*.

```
public void cmdReplace(PrintStream o, ArrayList opts) {
    Environment env = ((EnvWrapperT2)opts.get(0)).getEnv();
    if (env == null) {
        o.println("Error: the environment must be created first.");
        return;
    }
}
```

```

    if (opts.size() != 3) {
        o.println("Error: insufficient parameters given.");
        return;
    }

```

Similar to above, make sure the destination is valid.

```

    int numCells = ((EnvWrapperT2)opts.get(0)).getVarPopn();
    int destCell;
    try {
        destCell = Integer.parseInt((String)opts.get(1));
    } catch (NumberFormatException e) {
        o.println("Error: invalid cell number: "+e.getLocalizedMessage());
        return;
    }
    if (destCell < 0)
        destCell = numCells+destCell;
    if (destCell >= numCells) {
        o.println("Error: cell number out of range: "+destCell);
        return;
    }

```

Now get the key to retrieve it with.

```

    String key = (String)opts.get(2);

```

Now copy the cell in storage, and insert it into the environment.

```

    Cell c = (Cell)cellStorage.get(key);
    if (c==null) {
        o.println("Error: no cell in storage with key "+key);
        return;
    }
    env.setCell(destCell,c.copy(""),true);
}

```

C.1.16 cmdReset

Anything that needs resetting between runs or whatever can be done here.

```

public void cmdReset(PrintStream o, ArrayList opts) {
    numGens = 0;
}

```

C.1.17 byteToHex

This takes a byte, and outputs a two-character string that is that byte in hexadecimal.

```

public static String byteToHex(byte b) {
    char c;
    String out;
    char[] foo = {'0','1','2','3','4','5','6','7',
                  '8','9','A','B','C','D','E','F'};
    c = foo[b & 15];
    b = (byte)((b & 240) >> 4);
    out = "" + foo[b] + c;
    return out;
}

```

C.1.18 toDP

Java has an annoying bug when it comes to dealing with the rounding of numbers for display, in that they will end up a little bit out quite a few DP along, making it display things like 0.15625000000000006 instead of 0.15625. This function rounds to a specified number of DP in order to get rid of this, but still hopefully give the right number (so 0.47499999999 doesn't become 0.47 when it should be 0.48 (because it really should be 0.475)).

```
public static String toDP(int dp, double n) {
    String format1="0.";
    String format2="0.";
    for (int i=0;i<dp+4;i++)
        format1+="#";
    for (int i=0;i<dp;i++)
        format2+="#";
    DecimalFormat df1 = new DecimalFormat(format1);
    DecimalFormat df2 = new DecimalFormat(format2);
    return df2.format(Double.parseDouble(df1.format(n)));
}
```

C.2 EnvWrapperT2

This class provides a wrapper for the **EnvironmentT2** class, in order to store settings that are specific to the class. It provides accessors and modifiers necessary to move the information around. Most validity checking should be done by whatever is setting the values.

```
public class EnvWrapperT2 {
    private EnvironmentT2 env;
    private double varCosmic;
    private int varCrossrate;
    private int varPopn;

    public void setEnv(EnvironmentT2 e) { env = e; }
    public EnvironmentT2 getEnv() { return env; }

    public void setVarCosmic(double c) { varCosmic = c; }
    public double getVarCosmic() { return varCosmic; }

    public void setVarCrossrate(int c) { varCrossrate = c; }
    public int getVarCrossrate() { return varCrossrate; }

    public void setVarPopn(int p) { varPopn = p; }
    public int getVarPopn() { return varPopn; }

    public void create() {
        env = new EnvironmentT2(varPopn, varCosmic, varCrossrate);
    }
}
```


C.3 EnvironmentT2 class

This provides the environment class, specifically aimed at this task. It also keeps various statistics on the system.

```
import java.util.*;

public class EnvironmentT2 extends Environment {
    int[] [] convTable;
```

C.3.1 Constructor

This simply takes the three values and passes them to the superclass to deal with.

```
    public EnvironmentT2(int s, double r, int c) {
        super(s,r,c);
    }
```

C.3.2 initCells

Here the cells are initialised and inserted into the **ArrayList**.

```
    protected void initCells(int size) {
        cells = new ArrayList(s);
        for (int i=0; i<s; i++) {
            cells.add(new CellT2(getNextID(),new ArrayList(),0.0,
                                true,new GenomeT2()));
        }
    }
```

C.3.3 generation

This ensures that the fitness of all the organisms is recalculated at the start of a generation.

```
    public void generation() {
        for (int i=0; i<cells.size(); i++) {
            ((Cell)cells.get(i)).makeDirty();
        }
        super.generation();
    }
```

C.3.4 fitness

The most important method for this system is the one that determines the fitness. A lot of the description here is repeated from the description in Section C.1, however much is still in addition to it.

The *fitness* method is passed a **Cell** (Section B.2), from that it retrieves the **GenomeT2** (Section C.5). Then, the byte array corresponding to the actual organism is extracted. This is then passed through a simulation to determine what the fitness should be. The fitness determining test is done four times, each time starting in a different corner on an integer-graded plane. The starting values for each test are -5 and 5 in both x and y .¹ The fitness for each test is the change in Manhattan distance, i.e.

$$M = |x| + |y|$$

¹This means that the starting points used are: (5,5), (-5,-5), (-5,5), (5,-5).

so if it starts with a high distance, and ends with a low one, that corresponds to a high fitness.

Originally this used random starting points, but this has the problem that sometimes the same genotype could have a very different fitness value on two different fitness evaluations. It is interesting to note that it would still evolve the ideal solution, however made the results a little more messy than they needed to be.

The final fitness is the sum of the fitness of each of the four tests. This is then stored in the cell. While it is the case that the fitness may be different for every test, it is still used here to provide an easier way of generating statistics. The *generation* method will go through and mark all of the cells as 'dirty' at the start of each generation to account for this.

```
public double fitness(Cell c) {
    if (!c.dirty()) {
        return c.getFitness();
    }
    byte[] b = ((GenomeT2)c.getGenome()).getGene();
    int fit = 0;
    int[] xvals = {5,5,-5,-5};
    int[] yvals = {5,-5,5,-5};
    for (int i=0; i < 4; i++) {
        int x = xvals[i];
        int y = yvals[i];
        int st_dist = Math.abs(x) + Math.abs(y);
        for (int j=0; j < 8; j+=2) {
```

This first part does all the testing in the *x* direction.

```
        byte test = (byte)((b[j] & 240) >> 4);
        byte res = (byte)(b[j] & 15);
        if (test <= 4) {test = 0;} else
            if ((test > 4) && (test <= 9)) {test = 1;} else
                if (test > 9) {test = 2;}
        if (res <= 4) {res = 0;} else
            if ((res > 4) && (res <= 9)) {res = 1;} else
                if (res > 9) {res = 2;}

        if ((test == 0) && (x < 0)) {
            if (res == 0) {x--;} else
                if (res == 1) {x++;}
        } else if ((test == 1) && (x > 0)) {
            if (res == 0) {x--;} else
                if (res == 1) {x++;}
        } else if ((test == 2) && (x == 0)) {
            if (res == 0) {x--;} else
                if (res == 1) {x++;}
        }
    }
```

Now for the *y* direction...

```
        test = (byte)((b[j+1] & 240) >> 4);
        res = (byte)(b[j+1] & 15);
        if (test <= 4) {test = 0;} else
            if ((test > 4) && (test <= 9)) {test = 1;} else
                if (test > 9) {test = 2;}
        if (res <= 4) {res = 0;} else
            if ((res > 4) && (res <= 9)) {res = 1;} else
                if (res > 9) {res = 2;}
```

```

        if ((test == 0) && (y < 0)) {
            if (res == 0) {y--;} else
                if (res == 1) {y++;}
        } else if ((test == 1) && (y > 0)) {
            if (res == 0) {y--;} else
                if (res == 1) {y++;}
        } else if ((test == 2) && (y == 0)) {
            if (res == 0) {y--;} else
                if (res == 1) {y++;}
        }
    }
    if ((x != 0) && (y != 0)) { // from when random start
                                // points were used.
        fit += st_dist - (Math.abs(x) + Math.abs(y));
    } else
        fit += 4;
}
c.setFitness(fit);
return fit;
}

```

C.3.5 getCellConvergence

This implements the convergence calculation as detailed in the **Environment** class, Section B.1.11.

```

public double getCellConvergence(Cell c1) {
    CellT2 c = (CellT2)c1;

```

First a check is made to see if the lookup table needs to be built, and if it does, then we built it.

```

    if (convTable == null) {
        int s = cells.size();

```

For GAs of a non-fixed length, we will need an extra step here to determine the longest one, and make the table that size. In this case, each genome is comprised of 16 semantic units, and each of them can contain a range of 16 values.

```

        convTable = new int[16][16];
        for (int i=0; i<s; i++) {
            GenomeT2 g = (GenomeT2)((Cell)cells.get(i)).getGenome();
            byte[] dna = g.getGene();
            int t = dna.length*2;
            for (int j=0; j<t; j+=2) {
                convTable[j][(dna[j >> 1] & 240) >> 4]++;
                convTable[j+1][dna[j >> 1] & 15]++;
            }
        }
    }
}

```

Now that we know we have a valid table, we check the cell we have been given against it.

```

    byte[] dna = ((GenomeT2)c.getGenome()).getGene();
    int s = dna.length*2;
    int popn = cells.size();
    double count = 0.0;

```

```

    for (int i=0; i<s; i+=2) {
        count += (double)convTable[i][dna[i >> 1] & 240 >> 4] / popn;
        count += (double)convTable[i+1][dna[i >> 1] & 15] / popn;
    }
    return count/s;
}

```

C.3.6 clearCellConvergence

This resets the convergence cache.

```

public void clearConvergence() {
    convTable = null;
}

```

C.3.7 random

This simply gives a random number between two values.

```

public int random(int a, int b) {
    return (int)(Math.random()*(b-a+1)+a);
}
}

```

C.4 CellT2 class

This class contains the implementation-specific code to represent a cell. It provides a consistent communications channel between the genome and the environment. This has a variable that contains a reference to the DNA variable from the superclass, however it is of a different type. This allows the relevant one to be selected depending on what is necessary (i.e. the full range of methods for internal use, or a consistent set for passing around).

```
import java.util.*;

public class CellT2 extends Cell {

    protected CellT2(String i, ArrayList p, double f,
                     boolean d, Genome g) {
        super(i,p,g);
        fitness = f;
        dirtyBit = d;
    }
}
```

C.4.1 mutate

This method mutates a genome by simply changing a random byte to a random value.

```
public void mutate() {
    ((GenomeT2)DNA).putGene((int)random(8),(byte)random(256));
    dirtyBit = true;
}
```

C.4.2 crossover

This performs very simple one-point crossover between two cells.

```
public void crossover(Cell c) {
    byte[] b = new byte[8];
    GenomeT2 g = (GenomeT2)c.getGenome();
    int cp = (int)random(8);
    for (int i=0; i < cp; i++)
        b[i] = ((GenomeT2)DNA).getGene(i);
    for (int i=cp; i < 8; i++)
        b[i] = g.getGene(i);
    DNA = new GenomeT2(b);
    dirtyBit = true;
}
```

C.4.3 copy

This does a copy of the cell, including creating a new genome and so on.

```
public Cell copy(String newID) {
    return new CellT2(newID, (ArrayList)parents.clone(), fitness,
                     dirtyBit, DNA.copy());
}
```

C.4.4 random

This returns a double in the range $0 \leq x < r$

```
public double random(int r) {  
    return Math.random()*r;  
}  
  
}
```

C.5 GenomeT2

This class provides the domain-specific parts of the system. It stores the representation of the organism, and allows access to it. It also deals with anything to do with the structure of the data. In this example, it is simply an array of eight bytes. The interpretation of these is up to the **EnvironmentT2** class.

```
public class GenomeT2 extends Genome {
```

- *byte[] data*

The array where the data representing the organism is actually stored.

```
byte[] data = new byte[8];
```

C.5.1 Constructor

The size constructor is ignored, as this class requires fixed sizes. However, we add a default one that constructs a random organism, and one that accepts an array of bytes sized ≥ 8 , and we use the first 8 sections in that to start our organism with.

```
public GenomeT2() {
    randomise();
}

public GenomeT2(byte[] a) {
    if (a.length < 8) {
        System.err.println("GenomeT2(byte[]): array provided size "+
                           "must be  $\geq 8$ .");
        System.exit(1);
    }
    for (int i=0; i<8; i++)
        data[i] = a[i];
}
```

C.5.2 randomise

The starting point of a GA system is a collection of random algorithms. This kicks that off for each new instance of **GenomeT2**.

```
public void randomise() {
    for (int i=0; i<8; i++) {
        data[i] = (byte)(Math.random()*256);
    }
}
```

C.5.3 copy

In order to duplicate an organism so that it can be modified independantly, a deep copy must be performed. This ensures that actual duplicates of necessary data structures are made, and references aren't referring to the same objects.

```
public Genome copy() {
    return new GenomeT2(data);
}
```

C.5.4 getGene (one argument)

In order for the fitness of this genome to be evaluated, it must be able to be accessed externally. This provides that function, by returning the byte in array *data* at index *b*.

```
public byte getGene(int b) {  
    return data[b];  
}
```

C.5.5 getGene (no arguments)

Its a bit more useful and efficient to be able to get the data in one lump.

```
public byte[] getGene() {  
    return data;  
}
```

C.5.6 putGene

Somewhat the opposite of *getGene*, this allows a value to be placed into the genome at an arbitrary place. This allows things like external mutation to occur.

```
public void putGene(int b, byte val) {  
    data[b] = val;  
}  
}
```