EPCC-SS98-05

VisualGA: Visualisation of a Genetic Algorithm

Ruth Durie

Abstract

The project involved designing and implementing a visual simulation of a genetic algorithm (GA). This will be used in the first instance as a teaching aid for tutorials on genetic algorithms to facilitate understanding of how and why GAs work. It will include demonstrations of the basic methods GAs use, on a fairly simple problem.

This will be particularly useful in explaining the key concepts of GAs and to help to show how the method works.

Contents

1 Introduction

This project describes the attempt to build a suitable interface to demonstrate visually the progress of a Genetic Algorithm (GA). To do this, it must demonstrate the way characteristics in a GA spread, and graphically illustrate the main techniques a GA uses.

Previously, GAs have been taught at EPCC with the use of a GA interpreter, which printed text based results out at each stage to demonstrate changes in the GA on request. However, assimilating this output can be quite difficult - it can produce a great deal of text output very quickly. For instance, a very small run of a population of ten, for ten generations, will produce a hundred lines of (usually numerical) output. As the text is also output chronologically, it can also be difficult to separate out the different types of information (eg average fitness of the population versus the actual population itself).

Also, changes in a GA population tend to spread gradually - to see it working on a problem of any size, many generations may be required. So although text output suffices for a small population over a few generations, something better is required to really see the appropriate effects.

It would be considerably easier to illustrate changes if this output could be converted to a graphical display, presented in stages to allow changes over time. Several other basic GA concepts can also be usefully illustrated graphically, and possibly by using Java, these can be used in online teaching courses.

2 Background

2.1 Genetic Algorithms

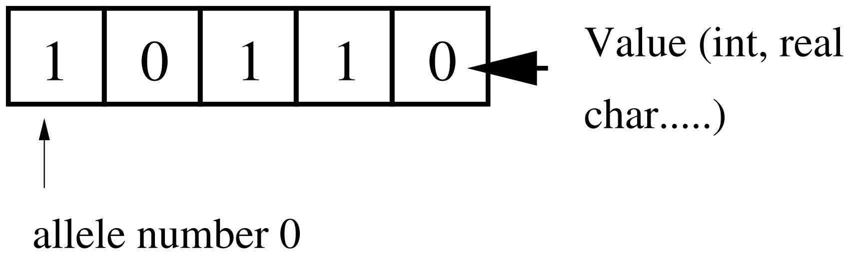
Genetic Algorithms are one of a group of problem solving techniques known as Evolutionary Strategies. They were first proposed by John Holland in the 1970s, as a general purpose method for tackling NP hard problems. Since then, his basic model has been expanded several times although the basic principles have always remained the same.

GAs are loosely designed to incorporate the principles of Darwinian evolution. They are used as a non domain specific heuristic to tackle NP-hard problems. While not a magic solution to all NP-hard problems [2], they have been used with impressive consequences on a wide range of problems. They have successfully tackled such diverse areas as network design, timetabling, bin packing and playing Tron [1].

Most well established problem solving techniques tend to consist of representing the problem in a certain form (eg a graph), and then using an established algorithm to work on the problem, or having one algorithm work with a set of parameters.

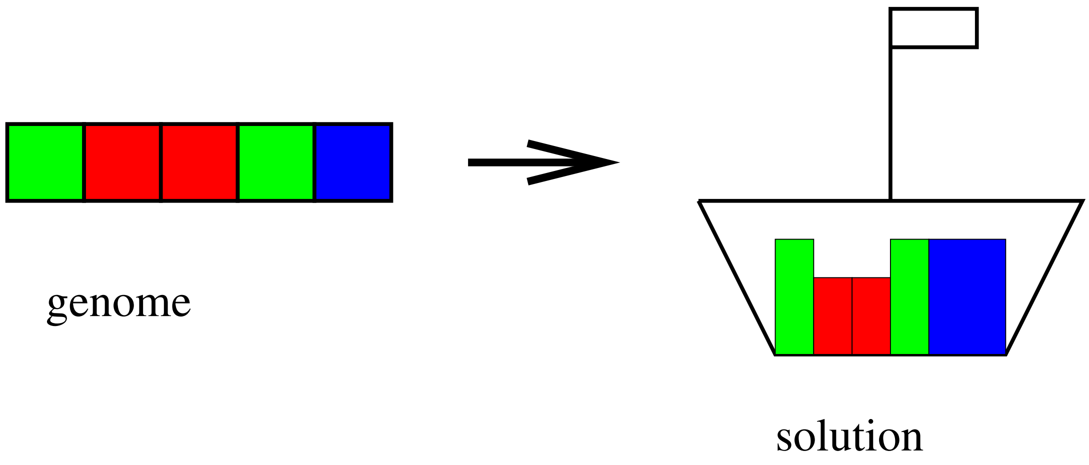
However GAs, rather than using one solution and trying to improve it, start with a population of many solutions, and try to produce better solutions from them. It works on the assumption that solutions with good characteristics will evaluate well, and spread. Each solution is encoded as a genome - a set of values (or alleles) for each part evaluation of the problem solution. For instance, one solution for bin-packing might have the first value set to the number of the first block to use, and so on.

Or , in the case of a function to maximise or minimise in N dimensions, an N length genome with dimension one's value in the first allele.



In general by making alterations to these genomes, better solutions may be produced over many iterations, provided a suitable representation can be found. There are three basic methods used to alter genomes - selection, mutation and crossover. The GA starts with an initial population (usually randomly generated), and then uses selection, crossover and mutation to produce two new children genomes from the initial population. It does this until it has enough children to form a new population, at which point the old generation is discarded and the new used to generate another population.

For instance, suppose the problem is to load a boat with different types of cargo, in such a way that the hold is not overfull but the maximum profit is made. The solution could then be represented by a genome with a list of numbers for each kind of cargo, the first number being the first container of the cargo to be loaded.



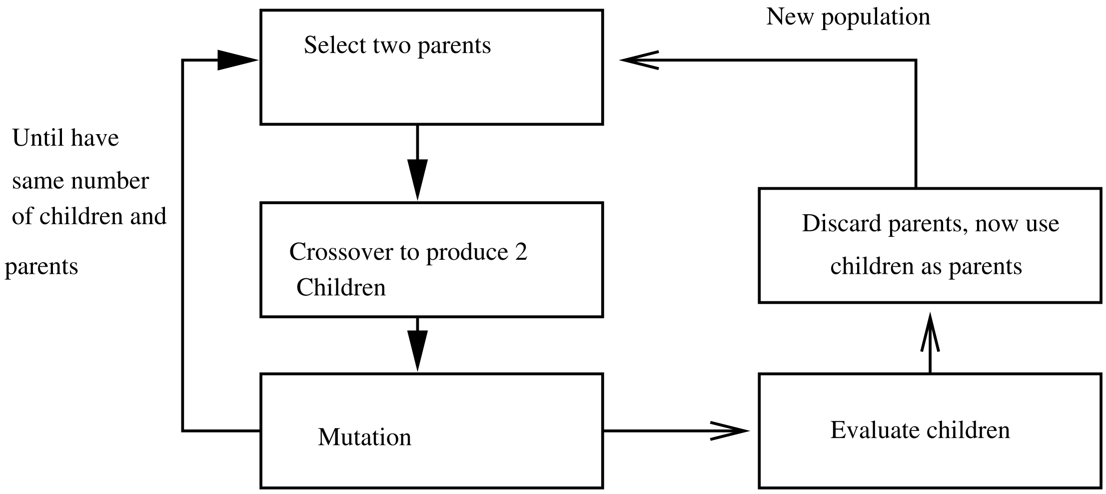
Each solution is rated for how well it solves the problem by a problem specific function, producing its fitness - a measure of how well it compares to all the other solutions. In the case of formulae, this would be what the equation evaluates to. With our boat loading example, it would be the amount of profit expected on the cargo. The GA will then attempt to minimise, or in this case maximise, the fitness. It is assumed the better the solution, the more likely it is to produce other better solutions. These fitnesses are used to guide which ones will be selected to either reproduce or go into the next generation. Literally, it is survival of the fittest.

2.1.1 Selection

Different types of selection include:

- **Roulette Wheel Selection** Imagine the genomes were organised into a pie chart, with each getting a chunk of the pie proportional to their fitness. Attach an arrow to the middle of the pie chart, spin it, and the section of the pie it stops in selects the genome it belongs to. Simply spin the arrow as many times as necessary to select as many genomes as wanted.
- **Tournament Selection** Genomes play a tournament against each other, with a set probability of the best in each round winning.
- **Stochastic Universal Selection** As roulette wheel, except instead of one pointer being spun N times, N pointers equally spaced are spun once. This is generally better than roulette wheel selection.

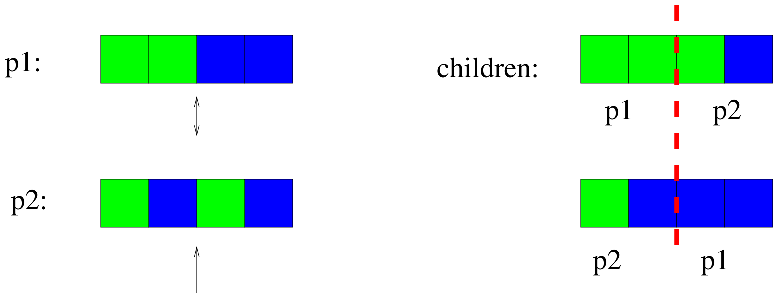
In Holland's canonical GA, the genomes were first randomly generated, and then evaluated, to produce the initial population. New populations were then produced by repeatedly using the processes of selection, crossover and mutation - based on their Darwinian equivalents - as shown below:



Once each genome has been evaluated, two must be selected for crossover. This can happen in many different ways. Quite often, the numerical evaluation is scaled to produce fitnesses so that no genome completely dominates the chances of being selected, or all the fitnesses being too closely spaced.

Selection is then done, usually according to fitness. In Holland's GA, the first genome was selected proportional to fitness, and the other selected randomly. It is now more common to select both parents with chance proportional to fitness.

2.1.2 Crossover



Crossover point (chosen randomly)

Crossover takes two selected genomes, and for each possible value in the two parent genomes, chooses a value to take for the child. In this way the child genome produced is a mixture of its two parents. With luck, it may pick the better value from each parent to produce an even better child. In most GAs, two children are produced, one the complement of the other, so that no part of the parents is lost completely.

Often crossover points are chosen for each pairing. For child one, everything up to crossover point one is copied from parent one. After the first point, and up to the next one, parent two is copied. After the third point, back to parent one. It is rather as if the two genomes had been laid out together on a chopping board, chopped into a number of pieces, and then every second piece swapped to produce two children.

In general, the more points used in crossover, the more disruptive crossover is. Uniform crossover has crossover points at every possible place and so breaks up adjacent alleles, but more commonly one point or two point crossover is used.

Genomes that are very fit may well be selected many times to use in crossover, and so the new generation will contain many products of a very fit genome, combined at different points with other genomes. In this way, by trying many different combinations with the fit genome, hopefully the GA will combine the good parts of separate solutions to make a better one.

2.1.3 Mutation

Mutation allows parts of the genome to be mutated, producing a new value which may contain something not in a previous population. This encourages diversity - if all the solutions currently in the population are very similar, the GA is unlikely to find a wildly differing solution. Mutation may introduce possibilities not in the initial population, or lost due to selection. Usually each value in a newly produced genome has a fairly low chance of being mutated, so only a few mutations will occur in the population as a whole.

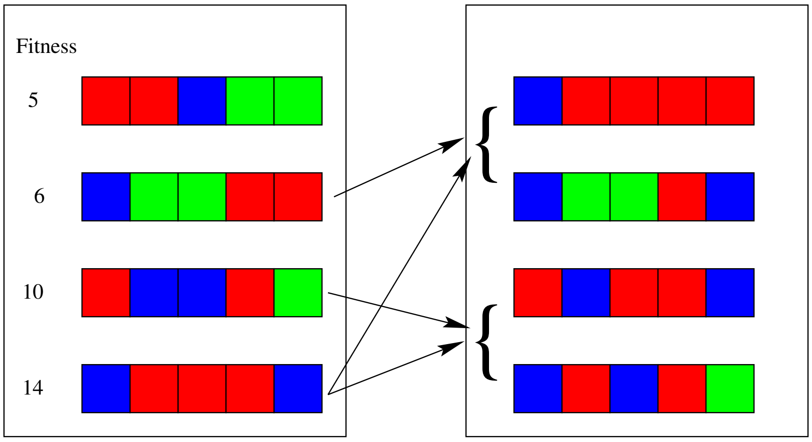
For instance, in the equation solving example, one of the values in the genome might have a percentage of its value added or subtracted. This could produce a value that did not exist at all at that position before.

Or, for the boat loading, one container of cargo might be swapped for a different type.

There are many variations of these basic versions of crossover and mutation to cope with different types of representations. For instance, when genomes represent block permutations for bin packing, or order of cities for the travelling salesman problem, the versions used above are unlikely to be of any help. Instead, crossover swaps the order of some values in the first genome according the some of the values in the second, producing a similar child. Mutation swaps a couple of cities randomly.

As long as the problem can be represented as a genome, operators can usually be invented to act as crossover and mutation operators.

2.1.4 The generational approach



Old Population New Population

In the generational approach once the two children have been generated by crossover and possibly mutated, they are inserted into a new generation. The GA then continues this process of selecting parents and generating children until the new generation is full. It then discards the old generation and starts using the new generation to produce another one. And so on.

The GA ceases to produce new populations when either the population has converged - all the genomes are the same , or a solution has been found that is good enough (ie has a certain fitness), or a certain number of new generations have been generated.

All these concepts are best seen visually - over time, the spread of certain combinations of values in the genomes can been seen distributing across the population, and the generation of new genomes is best explained by use of a new diagram. The population can be seen moving from a random initial one to once slowly converging and hopefully increasing in fitness. As fit genomes propagate parts through the population, these values can be seen spreading. Alleles may converge, as can be seen below.

The actual calculations for the GA were calculated using RPL2, a preexisting language at EPCC for designing a running genetic algorithms.

2.2 The Problem

Good problems for visualisation have to satisfy a number of requirements.

- It cannot be too difficult for the GA to solve with a small population. Simply, there are limits to how big the visualisation can be, and still fit on the screen satisfactorily.

- The representation of the problem as a genome should be fairly intuitive, so a non expert can look at the display and understand what the GA is doing, and have some idea in practical terms what is happening. It ought to be easy to construct the solution from the genome.
- For the same reason, the problem must be fairly easy to evaluate graphically, or else it may be hard for the user to follow how close the GA is to solving the problem. An intuitive evaluation function usually helps users to check they understand how the GA is converting genomes to solutions and evaluating them.

2.3 RPL2

Before going any further to describe the representation and form of the problem, it is helpful to describe the actual language used to build the GA. As RPL2's limitations, strengths and features did make a large difference to the design of the actual problem and the Java interface itself, it is worthwhile digressing to describe it here.

RPL2 is the language used at EPCC to create and control Genetic Algorithms. It consists if an interpreter, plus various libraries written in C. The actual GA is written down as a series of commands in a script, eg :

```
  numGen :=10;
  popSize:=6;

  for g := 1 to numGen
    ScaleRaw(current);
    Empty(next);
    currentBest := SelectScaledBest(current);

    for p := 1 to popSize
        mother := SelectScaledRoulette(current);
        father := SelectScaledRoulette(current);
        doCrossover;
        endfor
        Swap(current,next);
endfor
```

The built in libraries include genome representations, and a range of selection, crossover and mutation operators for alleles which are integers, reals, binary, permutation and sets. There are also some evaluation functions built in. Libraries are organised by type of alleles in the genome, eg the int library, and also contain their own functions for printing genomes etc. This later caused a problem as most **but not all** libraries printed out in the same format.

3 Method

The project splits into four parts, both chronologically and logically. The problems to use had to be selected, and then a GA written in RPL2. The actual Java interface for the visualisation needed to be written, and finally the interface and RPL2 had to be linked together. The sections below describe each of these stages in detail.

3.1 The Problems Chosen

Before designing the GA, it was worth working out which problems were likely to visualise well by meeting the criteria set out above. Two problems were tackled, scales balancing and the N Queens problem.

3.1.1 Scales Balancing

Scales balancing involved loading one side of a set of scales with weights, so that it balanced the other side perfectly. This problem also had an advantage in that it is useful practically, albeit in a more complex form. For instance in ship loading, the ship must be balanced properly and with the maximum profit possible. Another, more advanced version would have the GA loading both sides of the scales to balance.

The constraints for this problem are twofold - it is assumed only a certain number of weights can fit on the scales, and of course it must balance the other side.

The evaluation function is simple and intuitive - a measure of the absolute difference between the weights of the two sides, which the GA will then attempt to minimise. A later addition to the evaluation function adds in a small penalty for space wasted when stacking the scales.

The evaluation functions can then be drawn as set of scales for each genome.

3.1.2 N Queens Problem

The N queens problem was added at a later stage. It is a well known constraint satisfaction problem involving placing N queens on a NxN chess board so that none of the queens attack each other. This means three constraints : no queens on the same row, column or within a diagonal.

It was added mainly to demonstrate the use of the permutation representation, as many useful GA applications - for example scheduling and bin packing - use it. It shows that it is possible to visualise other problems in the interface without major alterations. It also has the advantage of demonstrating the GA approach to a problem used widely to test other algorithms.

Each genome can be displayed as a chessboard with the queens placed on. The evaluation function is then a count of how many times queens are under attack.

Both problems are solvable with small populations (ie under twenty genomes). Both have representations which can be converted with little effort to solutions, and can be evaluated graphically in obvious ways.

3.2 Representation of the problems in RPL2

RPL2 allows for extra code to be slotted in at different points: new representations can be defined, or simply new problems for the old representations.

3.2.1 Scales Balancing

The basic scales balancing problem was built as a representation dependent add on to the int library. A number of blocks with varying weights are provided in a separate file for the GA to use. The genome is a list of which blocks to put on the scale, with each block represented by an integer. Apart from being an easily understood representation, this also reuses code in RPL2. This also simplified the graphical display somewhat - integer numbers correspond simply to the colours used.

The integer representation provides a straightforward mapping between genome and solution. It is very similar to the boat example used in the first section, but with the space constraint involving scales space rather than hold space. Here weight is being maximised rather than profit, but the basic principle is similar.

The integer library also provided suitable crossover and mutation operators. The actual scales balancing library is called BalEx, and described fully in Appendix 1. It defines the evaluation functions, and a function to read in and set up the block weights.

A drawback to using this integer representation is that many genomes map to the same solution - the order the blocks are placed in the scales make no difference to the final solution. However, this does mean that smaller population sizes can be used, and as this is only a problem for demonstration purposes this is not a serious drawback. The GA can still find a satisfactory solution.

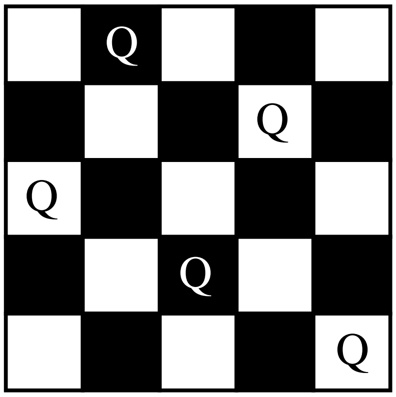
The fitness function used was the absolute difference between the weight needed and the actual weight. Although sets of weights that are too heavy can have the same fitness as those that are too light this seems to have little effect on the GA's ability to solve the problem.

However the interface did need to be able to distinguish between negative and positive weight differences. Otherwise it would not be possible to know which scale should be the heavier when drawing the evaluation function. Equally, it is quite helpful to know if the population consists of sets of blocks that are mostly too heavy or mostly too light.

Rather than altering the integer representation, which would involve writing many unnecessary functions, an extra allele was added. This is then set when fitness is evaluated to either 1 or 0 depending on if the blocks are too light or heavy.

Once the problem was working properly, a variation evaluation function was tried with a space constraint, to make the problem harder. This does work, but requires a bigger population and more generations. It is often helpful to people new to GAs to check what they think the genome should evaluate to against the number shown, and a space constraint makes it much more difficult to do so. However, both functions are provided and either may be used.

3.2.2 The N Queens Problem



The N queens problem was already provided in a library for the permutation library. The representation of this problem encodes the row and column constraints implicitly, saving the evaluation function from checking them.

If N Queens are placed on a NxN board, there must be one and only one per column or row. So let the alleles in the genome represent the columns from left to right. The value of the allele is then the row the queen is standing in. For instance, the picture above represents the genome [3 1 4 2 5].

So the genome is a N long permutation of the values 1 to N, as there must be a queen on each row. The row constraint and the column constraint need not be checked for violations by the evaluation function as these have been implicitly checked. It therefore checks the number of attacks on each queen via diagonals, and uses the total attacks as the fitness.

The permutation versions of crossover and mutation are used for this problem.

3.3 Design of the GA

GAs have a large number of design decisions associated. As explained earlier, selection can be done by roulette wheel, S.U.S or tournament selection (in which case tournament size and the probability of the best winning must also be set).Crossover and mutation rates can vary, and there is usually more than one operator available for either (eg depending on the number of points used for crossover). The GA can be steady state -inserting children straight back into the original population- or generational.

Most of the choices of how to build the GA are restricted by the library setup - eg the permutation library only has one type of crossover. Default scripts have been provided for both the problems that make choices that solve the problems in reasonable time. However, the user may substitute any script of their own, provided it obeys the following rules:

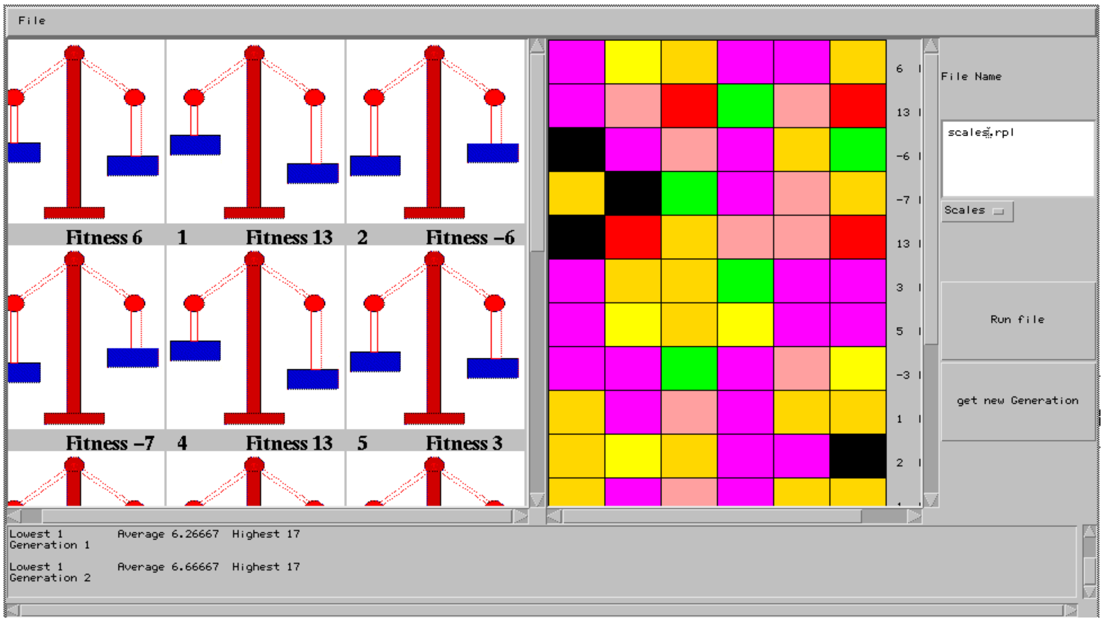
1. All scripts follow the generational approach, ie a new generation of children is generated to replace the parents completely..
2. It must, at each stage, write out all the genomes in the population so that the Interface can adapt.

3. There must be some output signalling the end of each generation. Just calling stats-format to print out statistics for the newly generated generation will do, or any printed text provided it does not start with '['

It was decided to deal with generational replacement, rather than the steady state scenario. This leads to a cleaner updating mechanism - the population changes all at once, rather than having to find out which genome has been replaced, and by what. Indeed, most functions in RPL2 provide no way of telling which genome has been replaced. It would be possible to adapt the Java interface to deal with this, by printing out the entire population when one genome is replaced, or after a certain number have been - this 'pseudo-generational' approach will work. However, there does not seem to be a way to communicate or find out that 'genome x has been replaced by genome y',.

The other decisions, ie population size and number of generations to run for, have been left to the user. While this could possibly be controlled by the interface, it would be quite complicated to do so - the link up method would need to be adapted, or more likely replaced. So instead this is left up to the original RPL2 script. This does have the advantage that users can make changes to the RPL2 script, and see these changes reflected in the speed and way patterns spread in the interface.

3.4 The Java Interface



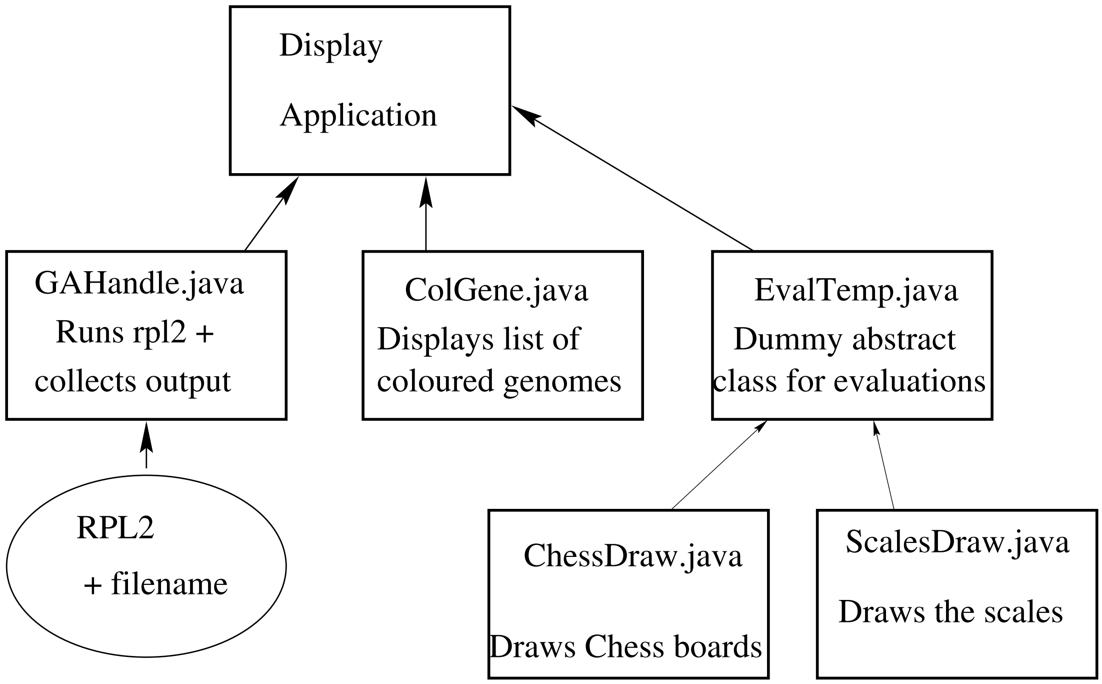
Main Interface Panel

Once the GA had been designed and was working satisfactorily, it was time to look at how to produce the visualisation. It was decided a user interface should be produced, and there were two systems available: X Windows or the Java GUI.

Java was chosen as it is the more portable of the two, and also easier to modularise for purposes of code reuse. Ironically the Java GUI toolkit calls the local GUI system to produce an interface - in the case of UNIX, it calls X Windows.

However, as running RPL2 requires a system call and a script name, there are likely to be security implications if the interface were embedded in a webpage. It may be possible, with some alteration, but it would very likely be unwise. So, the interface currently runs only as a Java Application.

The class structure of the interface has been designed so that classes can be reused elsewhere, and are as general as possible. Most classes take vectors (extendible arrays) of arrays as arguments. The class structure is as follows:



3.4.1 ColGene

ColGene produces a row of coloured blocks for each genome, and prints the fitness beside it. Note that this works with any problem where the genomes can be converted to an integer array. It will also change its preferred size according to how many genomes it has to display.

It would have been possible to have ColGene produce a canvas with only one genome on, and then take an integer array as an argument. This would have been good, as the class could have been overloaded to deal with different arrays apart from integers, eg chars. It would then automatically call the right methods depending on what type of array it had. Unfortunately, this would have resulted in creating a lot of objects for large populations, which seems to have speed implications for the interface.

A very similar class which takes strings as arguments is provided for the applet and discussed later.

3.4.2 GAHandle

GAHandle runs the RPL2 script, and collects the output. It also contains a function that will provide the next generation from the script when asked. This function actually returns not only the actual genomes, but also any other output in another vector. In the interface, this additional vector is copied to the status bar. This has the advantage that any errors, statistics reports or other output from the RPL2 script can be passed to the user. Vectors were used as they could be any size, and contain anything.

GAHandle also relies on the square brackets round the genome to identify it. This allows GAHandle to be used with most RPL2 scripts without the use of the debug library. Also, all scripts must have at least one line of non genome text between one generation and the next. Alternatively, if GAHandle knew the size of genomes expected, the population size and the format expected it in, it could possibly be adapted. But this would increase the amount of information the RPL2 script must convey before beginning its run. assuming the users may have many scripts they want to try, it makes more sense to assume most users will print out something (usually the statistics).

3.4.3 Display

Display contains a number of options. The 'west' panel consists of buttons, and a text field to set the script name. There is a list to allow the user to select the problem type. This passes responsibility entirely to the user for knowing what the problem is: the alternative would be to have GAHandle read in the problem name or search the vector for it. Every RPL2 script would have to print out the name of the problem.

3.4.4 Evaluation Function Visualisations

Only the actual evaluation function is problem specific. To make Display expandable in the future, and to provide a proper class structure, a dummy abstract class ClassTemp was created. This acts as a kind of template for the other classes actually implementing the evaluation function - they must overwrite the move method. This is implemented as a class rather than an interface so that objects can be declared as ClassTemp, loaded later with the class actually needed, and then the correct move method called for that class later.

Scales Problem

ScalesDraw is the class which provides and manipulates the scales image. It provides a method to move the scales the necessary positive or negative distance. Oddly, there was a problem with paint calls that necessitated threading this class (see actual code for explanation). Thus this class also has a finished function, to ensure the thread is stopped before the instance is destroyed.

N Queens Problem

The chess boards are drawn and manipulated by the ChessDraw class. This represents Queens by red squares, rather than letters, so that the boards can be shrunk as much as possible to allow as many as possible to be seen at once.

3.5 Linking the Two

The major problem was figuring out how to allow the Java interface and RPL2 program to communicate. Many options were investigated, with varying amounts of success. Initially the goal was to allow parameters to be passed to RPL2, as well as getting information from RPL2.

File Handling

Initially file handling was implemented, and while this worked, it was somewhat clumsy. There were also questions about file permissions, and stopping both the interface and RPL2 writing to the same file at the same time. While not insurmountable, these problems together with the messiness of the solution were enough to suggest looking for a different one.

Named Pipes

Named pipes would appear to be the ideal solution, although this would limit the interface to using UNIX. Named pipes allow two processes on the server to communicate with each other. Although some experimental C libraries were successfully written, there have been problems with synchronising programs without using the fork and exec commands. The C would also not port into Java, and the Java libraries as yet do not seem to support named pipes.

Most other methods suggested in text books involved programs in the same languages talking to each other, or via the Internet, but nothing on how two different programs on the same server may communicate. Trying to use the fork and exec command to set up a normal pipe seemed unlikely to work, as one part of the pipe would be written in Java and the other in C.

Sockets

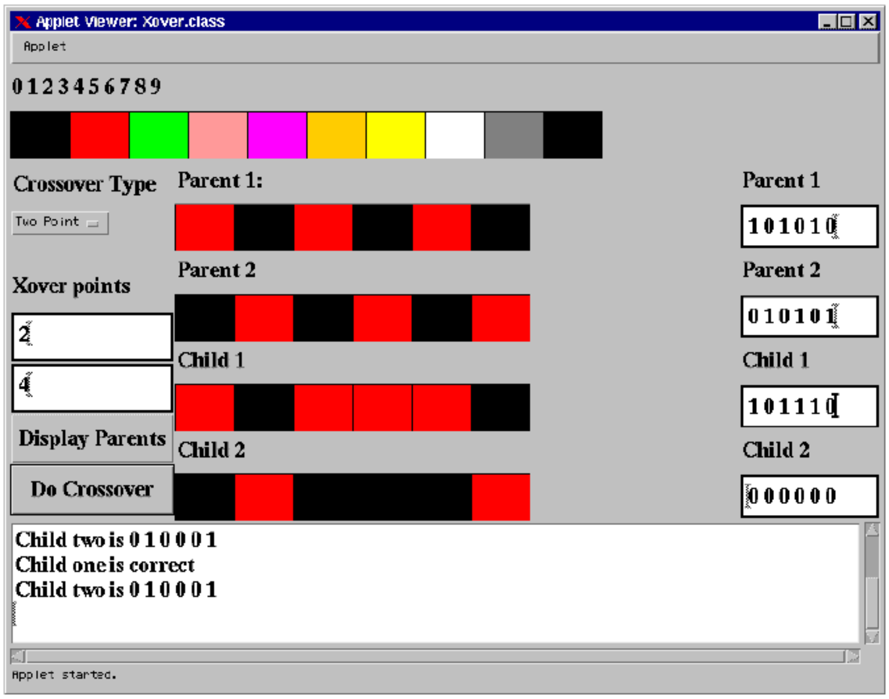
Another alternative was to use sockets. While this has its advantages - for instance, allowing the interface to monitor a GA running elsewhere, say on the Cray - this seemed somewhat over complex. I also suspected that while sockets are fairly easy to set up in Java, they could be very difficult to set up as a C library to add to RPL2.

Ultimately it was decided to have RPL2 write to standard output, and the interface read from standard input. This has some advantages - the Java is portable, and as long as RPL2 writes output in the standard format, could take output from RPL2 on different platforms. It does mean that the users cannot alter any of the parameters in the RPL2 script via the interface, although they can by use for a simple text editor. This was not one of the main goals of the project anyway, and so is not a serious drawback.

3.6 The Crossover Applet

The demonstration of crossover was done separately from the main interface, as to build it into the interface almost certainly would have resulted in an unacceptable level of screen cluttering, and diminished speed dramatically.

The applet currently looks like this:



The main class includes methods to calculate one point, two point and uniform crossover of two character arrays. It also sets up the display, and does most of the work.

A subsidiary class called GeneDraw draws one genome as a sequence of coloured blocks when given a string representing the genome.

4 Possible Expansions

4.1 Other Problems

To allow the interface to deal with all the rpl2 libraries, the following alterations would be needed. These would take a moderate amount of time.

At the moment the interface deals entirely with a vector of integer arrays. Display does not actually access the internals of the vector and so could simply be reused.

The GAHandle and ColGene classes would then be altered to deal with other sorts of input apart from integers. Reals would be the obvious one - at the moment Java will simply change reals to integers. GAHandle could parse the input into reals, or characters, after reading in the type of input from RPL2.

ColGene currently only uses an array of ten colour values to assign colour values to each allele. Possibly some sort of idea of range of the numbers could be used to assign colours, by breaking up the ranges into smaller sub ranges. If many values are possible, red green blue coding may have to be used to extend the number of colours available.

Characters could be converted to integers by transforming them to their ASCII equivalent, although again a greater colour range may be necessary.

Another problem is that PrintGenome prints out in different formats depending on the library used. All but the permutation library appear to print genomes out in brackets. It also needs there to be a space before the closing bracket. This could be patched by using more string processing, and is more of a nuisance than a serious drawback.

4.2 Controlling RPL2

This is more of a subsidiary concern, as the main concern of the project was visualising the GA. However, a greater degree of control over RPL2 scripts could be implemented.

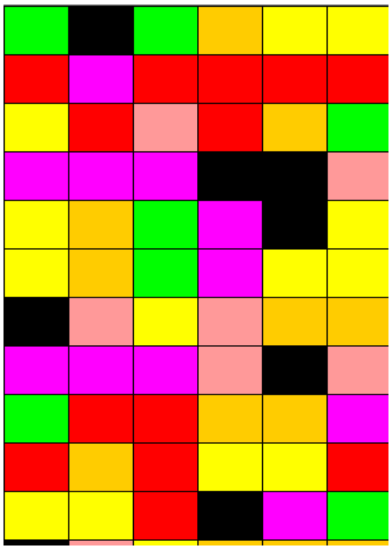
Having RPL2 manipulate the GA directly would require a better method of linking the two. Most of the other options for communication have been explored, although future releases of Java may introduce more options. C code can be ported into Java by using javah, but this misses the point of using Java.

A Perl controller could be used to modify the script automatically. It then could be called as a system call by the interface, with a number of command line options, or it could could set up communication via sockets. Sockets are probably over complex, unless the goal is to run the interface and RPL2 scripts on different machines.

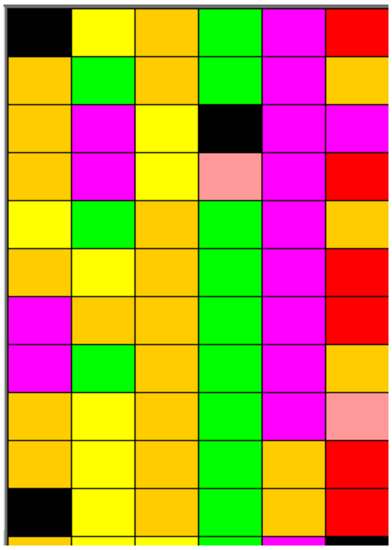
4.3 Other Facilities

Possibly a text editor could be built in, to allow the RPL2 scripts to be edited without use of a secondary editor.

5 Conclusion



Initial Random generation



Generation 28 - near convergance
on 3rd,4th and 5th values

Both the interface and the applet seem to work satisfactorily, although possibly more user feedback is needed to judge this. The interface does show patterns of genomes spreading, and so does demonstrate the GA searching the search space in a colourful way as can be seen above. The applet, being considerably more interactive, works well for teaching crossover. The ability to guess the answer would appear to be quite useful. Students can now work though the online course, and use the applet until they are happy that they have the appropriate grasp of crossover.

It was suggested that the interface could be used to control the GA, altering such parameters as mutation and crossover rate, and possibly even type of crossover used. While not a bad idea, this was deemed to be less about visualising the GA and more about having a GA Builder. It would have been interesting to develop one, but due to time constraints this never happened. It would be useful, partly because then the GA course could be taught without needing to teach RPL2, and no knowledge of RPL2 would be needed to write a GA. However, this looks like being more a project in its own right - perhaps one for next years SSP?

There were also some tradeoffs to be made involving speed and functionality. It would have been possible to integrate demonstrations of selection and crossover by having a separate popup display bring up the parents being selected, crossing over, and the children being put in the next generation. However, this would have caused the interface to become very large, and clutter up the screen. With a moderate sized population, it could also be very slow .

It was decided to have an interface capable of handling a very large population, as this way the visualisation could be better seen.

Overall, the visualisation is successful, although the question of demonstrating GA crossover has been left to a subsidiary applet. The code remains fairly easy to modify, and hopeful to reuse should other projects along these lines be proposed.

References

[1] http://dendrite.cs.brandeis.edu/tron/.

[2] W.G. Macready D.H. Wolpert. No Free Lunch Theorems for Search. Santa Fe Institute, February 1998. .

6 Appendix One: C Functions

6.1 In BalEx

BalSetUp Automatically reads in the weights and sizes from the filename specified when the library is requested. **BalEval(Genome,target weight)** Evaluates the genome incording to the target weight. **BaltoInter** Prints the genome out in the format the interface understands.

```
static char *sRCS_BALEX_C = "$Id: BalEx.c,v 1.3 1995/02/20 12:22:50 pds Exp $";

/* ******************************************************************** *
 *      Filename:       BalEx.c
 *
 *      Author:         Ruth Durie
 *
 *      Purpose:        Balance problem for Java simulator
 *
 * ******************************************************************** */

#include "BalEx.h"              /* operator prototypes as processed by rpp */
#include <repns/int/Int.h>
#include <math.h>



/*Global variables*/

int *weights;
int *size;
int room;
FILE *tovisual;

/*Read in the block weights and sizes from the file*/
BOOL BalSetUp(BOOL bUseNotDisamb,char *myfilename)
{
        int nBlocks,nCount;

        FILE *fp;
        if (bUseNotDisamb){
        nCount=0;


        fp = fopen(myfilename, "r");

        if (fp != NULL)
        {
        /*For some strange reason, won't scan "%d,%d", so I'm using 2
scanf stantements*/
```

```
            if (fscanf(fp, "%d", &nBlocks) == 1)
             {  fscanf(fp,"%d",&room);


                weights = (int *)MemAlloc(nBlocks*sizeof(int));
                size = (int *)MemAlloc(nBlocks*sizeof(int));
         while ( (fscanf(fp, "%d", &weights[nCount]) == 1)
                        && (fscanf(fp,"%d",&size[nCount])==1)
                                       && (weights[nCount] >0))
                {
                nCount++;
                }
            }

        }
        fclose(fp);
        }
        return TRUE;

}


/* ****************************************************************** */
/*The balance evaluation function- original, on weight only*/

void BalEval1(GENOME *pg,int target)
{
    INTG    *pvg;
    int     i,j,weight,final;



           if (pg == NULL) return;

              pvg = (INTG*)pg->pGenome;
              ErrAssert(NULL != pvg);
             weight=0;
              for (i=0;i<(nINTS-1);i++)
                {
                j=pvg->piAlleles[i];
                weight+=weights[j];
                }

                if (weight>target){
                        final=weight-target;
                        pvg->piAlleles[(nINTS-1)]=2;
                        }
```

```
                        else{
                        final=target-weight;
                        pvg->piAlleles[(nINTS-1)]=1;

                        }

                GSetRawFitness(pg,final);

}

/*Evaluation function, as follows:
Weight piles up until the scales are full. At that point, no more
weights count towards weight.
Some space penaty for wasting space

*/

void BalEval2(GENOME *pg,int target)
{
        INTG *pvg;
        int i,j,space,width,height,total,old,weight,final;

        total=sqrt(room);
        height=total;
        final=0;
        weight=0;
        old=0;

        if (pg ==NULL) return;
        space=0;
        width=0;
        pvg = (INTG*)pg->pGenome;
        ErrAssert(NULL != pvg);

        for (i=0;i<(nINTS-1);i++)
        {
          /*If there's still room left in the scales, add the weight*/
        if ((height !=0)){

          j=pvg->piAlleles[i];
          if (!(size[j]>total)){
             weight+=weights[j];
             old=width;
             width+=size[j];

          if (width==total) {
                height --;
                width=0;
```

```
            }
          if (width>total)
                {
                space+=(total-old);
                width=size[j];
                height--;
                }

          }
        }else {
          space+=5;
        }
        }
        if (height!=0){
                space+=(total-width);
                }
        if (weight>target){
                final=weight-target;
                pvg->piAlleles[(nINTS-1)]=2;

        }else{
                final=target-weight;
                pvg->piAlleles[(nINTS-1)]=1;

                }

        final+=(space*0.5);

        GSetRawFitness(pg,final);

}



/*Prints the genome out in the format the Interface understands*/

void BaltoInter(GENOME *pg)
{
        INTG  *pvg;
        int     i;
        char genome[99],fit[20];

          if (NULL == pg)
              {

        printf("NULL untyped genome\n");
         return;
              }
```

```
            pvg = (INTG *)(pg->pGenome);

            ErrAssert(pvg != NULL);

           printf("[");

           for (i = 0; i < (nINTS-1); i++)
                   {

            printf("%2d  ",pvg -> piAlleles[i]);

                   }

         if (pg->bEvaluated)
    {
        if ((pvg-> piAlleles[nINTS-1])==2)  /*Too heavy*/
        printf("] Fitness: %g\n", pg->rRawFitness);
        else
        printf("] Fitness: -%g\n", pg->rRawFitness);


    }
    else
    {
        printf(" ] <unevaluated>\n");
    }



}
```

6.2 In Inter

PermtoInter Function that overrides the normal print function in the permutation library, as it does not print out in the format expected by the interface.

```
static char *sRCS_INTER_C = "$Id: Inter.c,v 1.3 1995/02/20 12:22:50 pds Exp $";
/*It appears only to be the Permutation library that doesn't print brackets roun
genomes. Unfortunately, the Interpreter relies on this! Permutation is the only
library that doesn't, thus this print command just for it*/


#include "Inter.h"
#include <repns/permutation/Permutation.h>
```

```
/*Function to print genomes out in the format the interface expects*/

void toInter(GENOME *pg)
{
        PERMG  *pvg;
        int     i;


          if (NULL == pg)
              {
                 printf("NULL untyped genome\n");
                return;
              }

           printf("[");
        pvg=pg->pGenome;

        for(i = 0; i < pvg->length; i ++)
        {
                printf("%2d ", pvg->perm[i]);
         }
         if (pg->bEvaluated)
    {
        printf(" ] Fitness: %g\n", pg->rRawFitness);

    }
    else
    {
        printf(" ] <unevaluated>\n");
    }

}
```

6.3 Supervisors

- Kira Smyllie
- Roger Hare