A Report On Machine Learning And Genetic Programming Techniques, And Their Potential In Evolving Behavioural Animation Systems

Graham Brooks BACVA 3rd Year

Innovations

Contents

1.Introduction

My research for this unit is on the subject of Machine Learning, a topic which greatly interests me. Last year, my house-mate mentioned to me that on his course, they had been touching upon the idea that computer programs could, perhaps, write themselves (!). Computer programs that write other programs? Surely not, I hear you ask. Well, that's what I said to myself too, but for this year's Innovations project I decided that I would like to know what can be done…and maybe find out if computers really could do something extraordinary, like write their own programs.

My initial research led me to various interesting theses, dissertations, publications and documents, detailing a wide and wonderful collection of things that computers have already been found to be capable of. However, one area eventually seemed to appeal to me, given a background in Computer Graphics, and also a limited background in computer programming; Behavioural Animation Systems.

A project that I undertook last year involved the creation of a behavioural system which simulated the flocking/schooling characteristics of fish. I based the behaviours exhibited in the program on the Boids Algorithm (*Reynolds, C. W., 1987*), which describes a programming solution to the problem of entities in a flock detecting, and reacting to other members of the same flock around them. My implementation of this algorithm resulted in a convincing, if flawed, simulation of fish schooling. I wanted, as part of this unit, to discover ways of improving the fish schooling program that I had made using an innovative technique of some sort, which would optimize the behaviour of the fish.

It was about at this stage of my research that I came across the idea of Machine Learning. This is a rather broad term which covers many different techniques, although the basic principal is the same: a computer program can learn 'better' ways of carrying out a task. This in turn makes the assumption that the computer program knows what the task is, and more importantly, what 'better' is. If a program is provided with a task, a method by which it can judge its performance at that task, and a method for testing itself, it may, in fact, learn to optimise its performance at that task.

When I learnt of this, I was excited by the possibility that, perhaps, a behavioural system could teach itself to behave more realistically…literally evolve into a better, more realistic flock! This is the subject of my research as detailed in this essay; the implementation of a learning algorithm to optimise the performance of a behavioural animation system. The essay is also a survey of current techniques, documenting the course of my research into machine learning and genetic algorithms. Ultimately, I am aiming to answer the question of whether it is possible to apply Genetic Algorithmic learning to a behavioural system in order to improve the aesthetic behaviour of a flock or school.

2. Machine Learning – Designing A Learning System

In order to teach a machine to learn, we first need to define what it is to learn: we are, put simply, optimising our ability to carry out a task. When you learn to ride a bike, at first you fall off, but then you gradually get better until you are said to have 'learnt' to ride a bike; i.e. you have fully optimised your ability to carry out the task of riding the bike. We need to pose the problem of learning to a machine such that it can mathematically calculate whether it is improving or not, in relation to the task it is doing.

Let us consider the following definition:

> A computer program is said to learn from experience *E* with respect to some class of tasks *T* amd performance measure *P*, if its performance at tasks in *T*, as measured by *P*, improves with experience *E*.
>
> (*Mitchell T.M., Machine Learning, 1997*)

In other words, the program can be considered to have learnt if it gets better at carrying out a task following experience at attempting that task. It helps here if we begin to imagine the performance measure *P* as a numerical value, which, as the program learns, should improve.

This definition also points out some of the first important details into designing a learning system. The learning system must have the following components of design:

- **Task *T***: the task which we want the computer program to improve at/learn.
- **Performance Measure *P***: the value that determines the effectiveness of the programs current attempt at its task.
- **Training Experience *E***: the method by which the program test itself.

This definition enables us to specify, in a language the machine can understand, the parameters by which a learning task can be designed.

Let us look at the problem of learning how to play the game of backgammon. In this example, we use the above parameters to better specify our learning problem:

- **Task T:** playing backgammon
- **Performance Measure P:** percent of games won against opponent
- **Training Experience E:** playing practice games against itself

So, in this example, the machine learning how to play backgammon does so by playing test games against itself and rating its performance by whether it has won or lost. However, the problem is obviously more complex than this. In subsequent sections, I will continue to use the backgammon problem as an example.

2.1 The Training Experience

One critical choice in designing the learning system is deciding what type of training experience the program will have. There are two distinct types, primarily dependent on the task at hand:

Direct Training Examples

These would consist of actual direct information regarding the choice the program has to make each move of the game. For example, in backgammon, a direct approach would involve the program learning from specific board states and piece positions (i.e what to do in certain situations).

Indirect Training Information

The program may learn from indirect information, for example, by determining whether the backgammon game was eventually lost, it would then be able to infer that earlier moves were good/bad.

If the program is required to evaluate from indirect information, then we are faced with the problem of credit assignment. The program cannot truly infer that early moves in the game were bad simply because the game was ultimately lost. In actual fact, earlier moves may have been optimal up until a certain point where a mistake, or series of mistakes were made. Therefore, the program must determine on a move-by-move basis, what the state of the current board is, assign that a value which indicates how 'good' the board state is, and then evaluate what is the next best possible move. I will talk more about this later regarding the target function.

Another issue regards the level of interaction the user will have in the process of helping the program learn; whilst some learning could be totally automated and controlled by the program itself, this may not be ideal. For example, in CG, we may be developing a system which has to be aesthetically pleasing, and as such, we want human interaction to ensure that the result is appropriate.

In the case of a flocking system, the user may, for example, be asked to pick from a set of 100 test flocks, which appear visually the most pleasing. The program can then use the chosen ones to produce more of a similar trend of flocks. Again, the user can then select the most appropriate. The program is then learning to satisfy the user's requirements of the flock, rather than simply seeking the 'optimal' result; the problem with which is that the user may not have mathematically chosen a good optimisation function, as this can often prove difficult in complex problems.

2.2.1 The Performance Measure (Target Function)

The next thing to decide for our learning system is what exactly is to be learned by the system. The program for a game of backgammon would have, for example, a function called *Move*, which chooses the next move to make. What we are aiming to do is have the program learn how to make the *best* move every time.

How is the program to know what will be the best move for any given board state? It is evident that it is going to be impossible for the program to learn the function Move, simply because of the indirect experience available to our system. The program can, at the moment, only evaluate its performance based on whether the game is won or lost, therefore, deciding the best move at any given point in the game would require searching ahead through all of the possible moves to decide whether the game is won or lost. Obviously, this is unacceptable.

A better solution would be to produce a function which assigned a value to any given state of the backgammon board. Using the following notation, we denote that:

Target Function T : B > R

This implies that the target function we have chosen accepts as its argument any board state and produces as output a real number. This real number will denote the 'fitness' of the board (i.e. a rating of how good the board state is for the goal of winning the game).

Using this solution, it will be easy for the program to calculate only subsequent board states from the limited set of legal moves available, and select the board state with the highest fitness value. In this way, the system should select the best move available to it every time. This function is actually a representation of the original target function of *Move*; we have to make a representation simply because the target function itself is not efficiently computable. It would take far too long to evaluate whether each move is a good one depending on whether the game was won or lost, because the problem is recursive. Each move poses the same problem and so the possibilities are astronomical.

The process of finding an approximation to the target function is a key design choice in itself.

2.2.2 Choosing A Representation Of The Target Function

The decision to represent the target function with an approximation to it is a critical design choice, one which can be decided using a large number of techniques. To choose, we can first specify a number of attributes of a board state which would contribute to assigning a 'fitness' value to that state.

For example:

x_1: the number of black pieces in play
x2: the number of white pieces in play
x3: the number of black pieces in the home board
x4: the number of white pieces in the home board
x5: the number of black pieces beared off
x6: the number of white pieces beared off

All of these attributes can be used together to determine a fitness value for a board state by calculating in a linear fashion. Using the values given by the above variables and combining them with a weighting which denote importance of the attributes, a board state can be given a 'fitness value'.

i.e.

$$\check{T}(b) = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6$$

Where $\check{T}$ is the target function representation, b is any given board state, and where w_0 through w_6 are numerical coefficients. These values are quite simply weights which may vary throughout the programs learning of the game. The programs new learning task is to assign values to these weight variables such that the approximate target function performs suitably.

In summary, the design of the backgammon learning system up to now is as follows:

- Task T: playing backgammon
- Performance measure P: percent of games won against itself
- Training experience E: games played against itself
- Target function: T: B > R
- Target function approximation

  $$\check{T}(b) = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6$$

2.2.3 Function Approximation Algorithm

We now need a set of examples from training which denote the training values for specific board states – a starting point if you will. The states that we know about up to now include that if either side has already won the game. If we say that the training value in this case will be either +100 or -100 (depending if the game has been won or lost), we can construct a pair of values in the following form: $\{b, T_{train}(b)\}$. These pairs represent b, the board state as specified by the target function approximation $\check{T}$, and $T_{train}(b)$, the training value for that board state.

For example, in the case that black has won the game:

{{x1 = 0, x2 = 5, x3 = 0, x4 = 5, x5 = 15, x6 = 1}, +100}

2.2.4 Training Values

The only information as yet available to our learner is whether the game is eventually won or lost, but we require that specific 'scores' be assigned to individual board states at intermediate levels of the game. We cannot do this at the moment since we have no way of assigning the weights to the $\check{T}(b)$ target function approximation function.

As previously mentioned, just because the game was eventually won or lost does not mean that earlier stages of the game were played well or badly in light of that. If the program ends up loosing the game, it may still be the case that earlier moves played were optimal and that only one bad move resulted in the loss of the game. In which case, we want to know that was what occurred, so that the program still learns from the good moves as well as the bad ones.

One approach described by *Mitchell (1997),* is to assign the training value $T_{train}(b)$ for any board state b to be $\check{T}$(*successor*(b)), where $\check{T}$ is the learner's current approximation to the target function T, and where successor(b) denotes the next board state for which it is the program's turn again. In other words, we use the current approximation for $\check{T}$ as a base to test the next move.

This will make sense when you consider that we have already established that $\check{T}$ is a more accurate approximation to the target function at the end of the game, when we know for sure what the value of the board state will be.

3. Genetic Algorithms

Genetic algorithms were developed by John Holland, colleagues and students at the University of Michigan. The algorithms apply methods observed in the natural evolution of nature to computer programs, such that some of the qualities of natural selection may be recreated.

3.1 Comparisons to Natural Selection And Evolution

Genetic Algorithms draw their main inspiration in search optimisation from the ideas put forward by Charles Darwin about natural selection and evolution. These processes might indeed be described as nature's own search optimisation algorithm: the better, fitter organisms in a population are chosen to survive, and thus procreate, producing the next population, or generation.

In the analogy provided in the previous chapter, we could classify nature as if it were a computer program for which we were designing the learning algorithm:

- Task T: create a population that can survive
- Performance Measure P: number of offspring successfully created
- Training Experience E: life?

The analogy can then be made, if we consider nature to be a program, and the organisms within nature to be the hypotheses, that nature learns to select only the fittest organisms (hypotheses) to survive. These organisms then procreate to produce the next population, which by definition of survival, are more optimal organisms at surviving.

3.2 Why Use Genetic Algorithms?

One of the main reasons for using a Genetic Algorithm is because of its robustness as a method of search optimisation over large search spaces. In comparison to other methods, GAs are more efficient yet also more rigorous in their search method. This is highlighted by David Goldberg (*Genetic Algorithms in Search, Optimization And Machine Learning, 1989, p2*):

> "…where robust performance is desired (and where is it not?), nature does it better; the secrets of adaptation and survival are best learned from the careful study of biological example…Genetic algorithms are theoretically and empirically proven to provide robust search in complex spaces."

To explain this, I will observe some other traditional search methods, and compare their robustness (i.e the completeness of their search of the population), with that of GAs.

Calculus provides us with one way of searching a population of hypotheses in order to find maximum or minimum points. There are two methods of similar approach. One of these finds a local extrema (maximum point) by searching the space for gradients of 0, using the derivative of the curve. The other finds the optima by searching local gradients and 'climbing' the hill to find the highest point on the curve.

Both of these methods however are local in their scope; they both depend on the position of the initial point in the search initialisation. If we compare Fig 3.1 and 3.2 below, we can observe that in the local search area, the maxima would be found, however, a wider search reveals that the bigger peak was missed, and as such this is a massive shortcoming of the search algorithm.

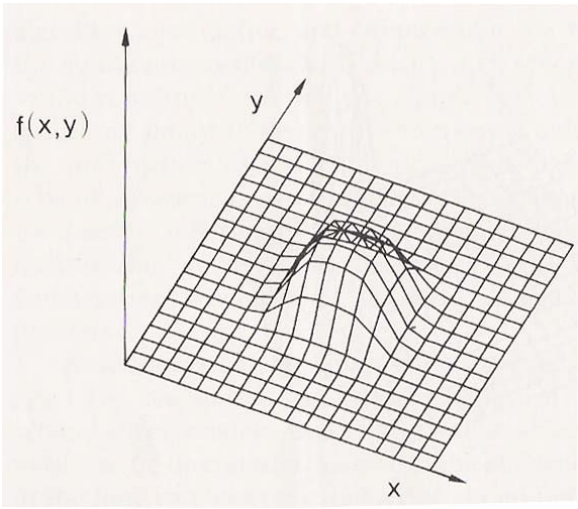


Fig 3.1

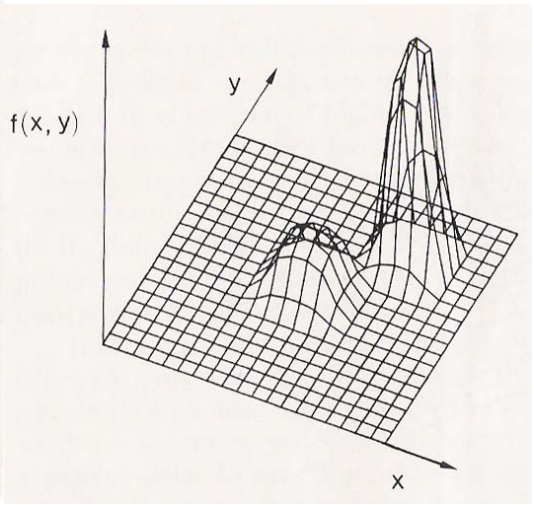


Fig 3.2

(*Genetic Algorithms in Search, Optimization & Machine Learning, by D. E. Goldberg, p 3*)

Also, calculus methods depend on the search space being defined by neat mathematical functions on which differentials can be found. However, this may not be the case. Although more complex curves may be approximated into a mathematical function, this in itself becomes inaccurate as a search method.

Enumerative search methods, which involve searching the space by analysing objective function values one at a time must be discounted for simple practical reasons. Whilst the robustness of these systems cannot be denied, their efficiency over larger or more complex search spaces make it unsuitable for anything but the smallest and simplest problems.

Genetic Algorithms use random choice as a tool in a very directed search process. By randomly selecting hypotheses in the search space, and then comparing them to a fitness function, it can traverse and define the population very quickly and yet almost exhaustively.

3.3 How Do Genetic Algorithms Work?

Genetic Algorithms search across a collection of candidate 'hypotheses' in order to identify the best hypothesis; in other words, that hypothesis that best matches that which is defined by the 'fitness function'. We can compare these terms to the Chapter 2 definition, in which 'hypothesis' would be analogous to the test data, and the 'fitness function' is equivalent to the target function.

The pool of different hypotheses generated as test data by the program is called a 'population'. Genetic Algorithms work by iteratively updating the population according to their level of 'fitness'. On each iteration, each hypothesis is evaluated according to the fitness function, and then the next population is created by probabilistically selecting the most fit candidates from the current population. However, the method by which the hypotheses pass into the next population differs; some will pass through intact, whilst others will have genetic operators applied to them such as crossover or mutation.

An example of the pseudocode for a Genetic Algorithm is detailed below:

GA(*Fitness*, *Fitness_threshold*, *p*, *r*, *m*)
Fitness: *A function that assigns an evaluation score, given a hypotheses*.
Fitness_threshold: *A threshold specifying the termination criterion.*
p: *The number of hypotheses to be included in the population.*
r: The fraction of the population to be replaced by Crossover at each step.
m: The mutation rate.

- *Initialise the population*: $P \leftarrow$ Generate p hypotheses at random
- *Evaluate:* For each h in P, compute *Fitness(h)*
- While [max *Fitness(h)*] < *Fitness_threshold* do

  Create a new generation, P_s:
  1. Select: Probabilistically select $(1 - r)p$ members of P to add to P_s. The probability $\Pr(h_i)$ of selecting hypothesis h_i from P is given by

     $$\Pr(h_i) = Fitness(h_i) / \Sigma^{p}_{j=1} Fitness(h_j)$$

  2. Crossover: Probabilistically select $\frac{1}{2}(r.p)$ pairs of hypotheses from P, according to $\Pr(h_i)$ given above. For each pair, $\{h_1, h_2\}$, produce two offspring by applying the crossover operator. Add all offspring to P_s.
  3. Mutate: Choose m percent of the members of P_s with uniform probability. For each, invert one randomly selected bit in its representation.
  4. Update: $P \leftarrow P_s$.
  5. Evaluate: For each hypothesis h in P, compute *Fitness(h)*.
- Return the hypothesis from P that has the highest fitness.

(Mitchell T.M, Machine Learning, 1997)

This algorithm creates a population of initially random hypotheses, each of which is evaluated against the fitness function to give a fitness value for the hypothesis. The function max Fitness will simply find the current hypothesis with the highest fitness

value in order to compare it to the fitness threshold. If it is below this value, the GA will continue operating on the population.
The first stage of the GA itself is to select members of the population for 'survival' – i.e. passing them into the next population P_s. It does this using a function of probability, where the likelihood of any one hypothesis being selected is proportional to its own fitness value, but inversely proportional to the fitness of competing hypotheses. In other words, those hypotheses with higher fitness values are more likely to be chosen for 'reproduction'.

Next, the algorithm selects half of the number of pairs of hypotheses chosen for the first step in order to apply the crossover operation to them (the crossover and mutation techniques are described in more detail in subsequent chapters). It then adds the 'children' hypotheses generated by this operation to the next population.

The final alteration to the new population is to mutate a percentage of them. This simply involves randomly selecting one of the bits of the hypothesis representation, and flipping it around. This simulates random mutation in natural states of reproduction.

The algorithm then finally updates the new population to be the current population, and evaluates the fitness of the new hypotheses according to the fitness function.

3.3.1 Hypothesis Representation

As I have explained, genetic algorithms apply operations to hypotheses on the basis that they are represented by bit-strings, or strings of 1's and 0's. So, the question arises, how does one represent the hypotheses as a string such as this? In the example of a flocking system, hypotheses might be presented as a series of parameters regarding, for example; the speed of the entities, their field of vision, etc. These parameters describe variables that control the flock, but would probably be in the form of vectors or angles. In order to implement the genetic algorithm, we need to convert the parameters of the flock into a series of bit strings.

For example, we might consider that the field of vision angle is a floating point variable between 0 and 360 degrees. In order to convert this into binary form (base 2), the variable must first be made discrete; that is, separated by clear, consistent intervals. The logical choice for this example would be to separate into 1 degree intervals. However, using bit representation, a more logical choice would be to use 8 bits, since this can represent base 10 numbers up to 255. Of course, if we are to use an 8 bit number, we have to divide 360 degrees by 255 to find the degree separation that each bit represents. In this case, a change by 1 bit would give a value of 360/256 = 1.40625.

To determine our choice we have to consider the necessary accuracy of individual parameters in the expression. If we decided for example that a 1.4 degree separation was not accurate enough, there are various solutions to the problem. We could add an extra bit to the string, in which case the string can represent numbers up to 511 (512

including 0). Therefore, the degree of separation given by using 9 bits would be 360/511 = 0.703125. It can be observed that by increasing the bit length by 1 bit, the accuracy can be doubled.

If there were no other option than to represent a floating point number for reasons of accuracy, then bits can be used to denote the number either side of a decimal point. This can be observed below:

2^x	128	64	32	16	8	4	2	1	.	½	¼
String	0	0	1	0	1	0	0	1	.	1	1

This string represents the base 10 number 41.75.

Choices must be made dependent on the parameters that are being dealt with when it comes to encoding them in binary representations. If it is absolutely necessary to encode the binary such that a high level of accuracy is maintained, then there are means to do so. However, thanks to the robust nature of Genetic Algorithms, this is rarely necessary.

There are some in-depth techniques regarding the use of building-block theories and schemata in order to design the coding, but these are very specialised and could require a project in themselves to cover in enough detail to understand them. I will note however that these techniques are designed to assist the process of Genetic Algorithm searching by clever design of the coding such that the crossover and mutation processes do not disturb collections of useful bit strings.

3.3.2 Crossover Operation

The crossover operation is one of the aspects of genetic programming which simulates the way in which nature produces variation in the genetic make-up of children to its parents. In the natural process of sexual reproduction, the child's genetic construction is that of a combination of it's two parents DNA strings.

In genetic algorithms, this effect is reproduced with the crossover operation, which simply selects sections of each parents bit-strings to swap in order to construct the children bit-strings. For example:

Parent 1: 0 1 0 1 ¦1 0
Parent 2: 0 1 1 0 ¦0 1

The crossover point may be generated randomly, to choose the point at which the strings will be crossed, in this case represented by the symbol (¦). Here, crossover has been chosen at position k = 4, therefore, the resulting strings will be:

Child 1: 0 1 0 1 0 1
Child 2: 0 1 1 0 1 0

As you can see above, the crossover operation, put simply, just swaps over the bits after the crossover point, such that the resulting children consist of a part of the original, and a part of the other parent bit string. In this fashion, resulting children will be partly genetically identical to their parents, but partly different. The idea of this process is to introduce variation to the offspring, but such that the variation avoids being totally random.

Crossover may also take place at more than one position in the bit string. In a two-point crossover, the middle of one string is replaced by the middle of a second string, and vice-versa.
e.g.
P_1 = 0 1 | 0 1 1 | 0
P_2 = 0 1 | 1 0 0 | 1

C_1 = 0 1 1 0 0 0
C_2 = 0 1 0 1 1 1

The crossover operation may indeed be carried out at any number of positions throughout the bit string. The crossover itself may then be described by a 'crossover mask'. This is simply another string which defines which bits should be crossed between two parent strings, and which ones should remain the same. In the above example, the crossover mask would be { 0 0 1 1 1 0 }, which details the fact that three digits should be swapped.
The following diagram taken from *Machine Learning*, demonstrates the crossover operation and the crossover mask:

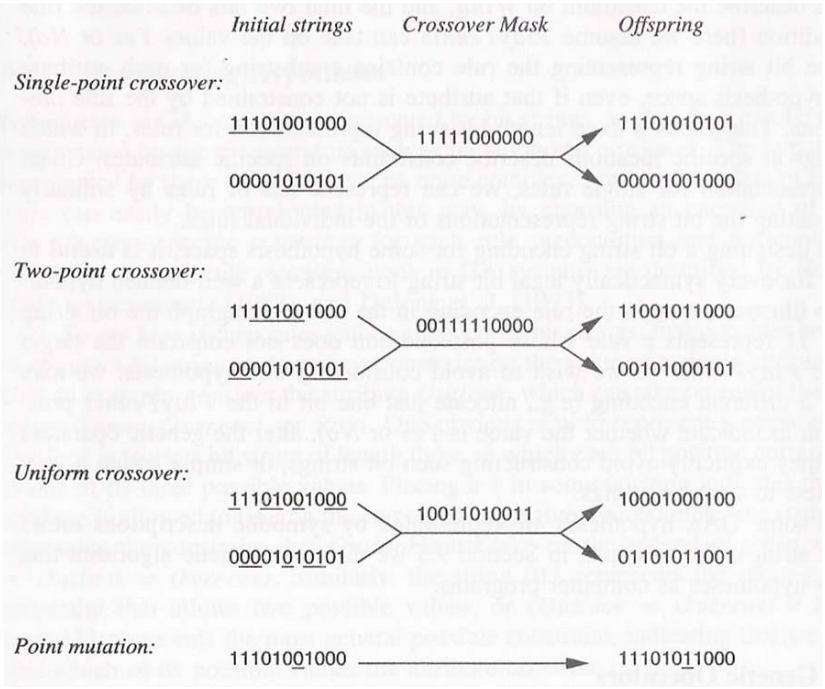


Fig 3.3 *(Machine Learning p 254)*

3.3.3 Mutation

Diagram 3.3a also details the operation of mutation; another aspect of Genetic algorithms which introduce variation to each generation of populations. This operation simulates the similar natural process of mutation in which random changes to the genetic make-up of offspring occurs which do not relate to the parent's DNA. These mutations often provide species in nature with a random evolutionary edge over their competitors.

The operation simply flips a single bit within an hypotheses at random. This operation takes place on just a single bit string in this case; it is not crossed with another bit string. For example:

P_1 = 0 1 1 <u>1</u> 0 1
C_1 = 0 1 1 <u>0</u> 0 1

In this case, the randomly selected bit to flip is k = 4, so in the child string, this bit is changed from a 1 to a 0.

Again, the operation may be described using a mask, which simply uses a 1 to describe which bit of the string should be swapped. For the above example; **0 0 0 1 0 0**

3.3.4 Selection Using The Fitness Function

The technique used to select hypotheses from the population according to their fitness level, as determined by the fitness function, is sometimes called the roulette wheel selection. This is because, the method of selection determined by the equation

$$\mathbf{Pr}(h_i) = Fitness(h_i) / \Sigma^{p}_{j=1} Fitness(h_j)$$

works probabilistically, in much the same way as the roulette wheel does. The probability of any one hypothesis being selected is proportional to its own fitness, whilst also being inversely proportional to the fitness of other hypotheses in the population. This is modelled accurately on the nature example itself, since this describes the competition to survive in life. 'The survival of the fittest' idea is wholly represented by this selection algorithm, except that it also takes into account the element of random chance not allowed for by the statement on its own.

In nature also, the likelihood of an organisms survival is proportional to its own ability to survive, but also inversely proportional to other organism's ability to survive. This describes the notion of competition in the population.

4. The Boids Algorithm (Craig Reynolds, 1987)

Since I am ultimately hoping to design a flocking system that uses a genetic algorithm to learn how to flock, I will need some basis of understanding how flocking systems work. Fortunately, I have had contact with the algorithm developed by Craig Reynolds before, and have made a working example myself.

However, I chose to rekindle my research of the algorithm for this project, in the hope of regaining my understanding of it and providing myself with an idea of how to implement the learning algorithm. In my research of Boids, I was hoping to discover traits that would allow me to apply a genetic algorithm, such as parameters that can be learned, whether there has been any previous research into the idea of applying genetic operators into flocking systems, and whether the idea of implementing a learning algorithm is either feasible or necessary.

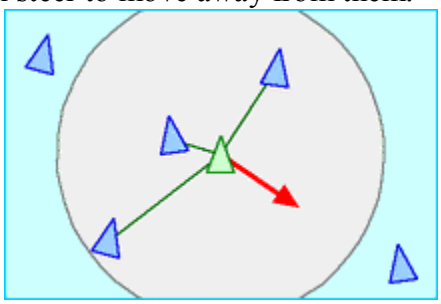
I will first give a brief background of the Boids Algorithm that I have researched, in order to provide the essential points to understanding my later proposed extension of the algorithm. Then I will answer the latter question above with a look into the research of *Zaera, Cliff, and Bruten, 1996*, in which they make an interesting observation regarding the use of learning algorithms and genetic operators, and whether they are too time consuming to design to outweigh the time required to manually design the flock behaviours.

4.1 How Does The Boids Algorithm Work?

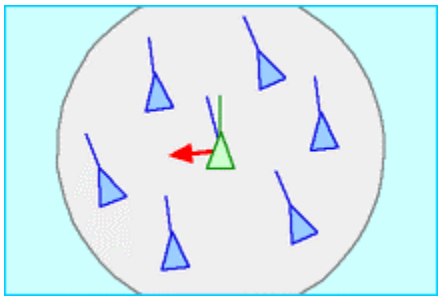
In 1987, Craig Reynolds developed an algorithm which described the way individual entities in a flock recognise, react to, and indeed, flock with other entities around them. This ultimately describes the motion of flocks in natural models such as birds or fish.

Reynolds calls the individual entities in a flock Boids. The algorithm basically specifies that the behaviour of a flock is driven by three primary behaviours:

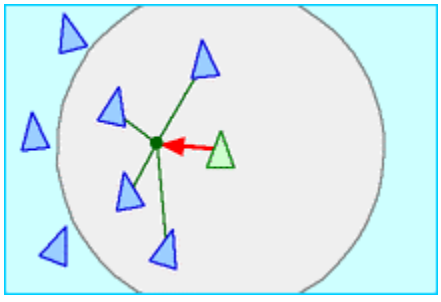
- Separation – Boids must have a behaviour which drives them to maintain a certain distance between local flock-mates. If they find themselves too close to a local flock-mate, they will steer to move away from them.



- Alignment – Boids have a tendency to move in the same direction (a general specification of a flock). This behaviour makes Boids aim towards the average heading of local flock-mates.



- Cohesion – Boids must also have a behaviour which keeps them close together. The cohesion behaviour makes Boids steer towards the average position of local flock-mates.



These three behaviours together basically determine the way that the flock works, since their weighting and values describe how each one is prioritised over another. These rules are listed in basic order of priority, such that boids will always aim to separate before they try to align. If this were not the case, the separation rule would be rendered obsolete, since the boids would simply align no matter how close they were.

The references above to 'local' flock-mates refers to the theory in Reynolds' Algorithm that each Boid has a local space in which it can 'see' other Boids. This would for example simulate visibility conditions in air/water as appropriate. (see Fig 4.1) [1]

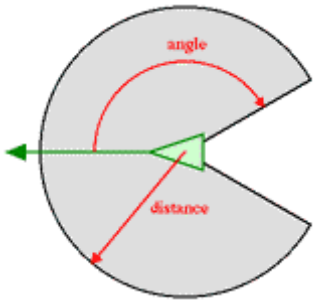


Fig 4.1

This diagram shows a local area around a Boid in which the Boid would be able to recognise any other local Boids, and react accordingly. Any Boids outside of this 'area of influence would be ignored.

[1] Taken from http://www.red3d.com/cwr/boids/. Craig Reynolds' website detailing the Boids Algorithm.

4.2 My Interpretation

My own implementation of the Boids Algorithm is a slightly different approach to that detailed by Reynolds in his '87 paper. I argue that rather than go through the difficulty of getting Boids to react to all of the local flock-mates, why not get each Boid to react only to the member of the flock they are closest to. I basically deduced that this should work since Reynolds' algorithm specifies that the most important rule is that of avoidance (separation). If this is to be the case, the nearest neighbouring Boid is the only one that should matter, since at any given time, it is the closest to collision. If a Boid should then continue on its path and find that another neighbouring Boid is suddenly closer, then it would choose to avoid that one instead.

I believe this makes logical sense in terms of the behaviour of a flock; if each Boid is effectively only concerned with its closest neighbour, the recursive nature of the interaction of the Boids throughout the flock will still ensure that every move is reflected in the flock as a whole.

Consider, for example, a Boid in the centre of the flock moves to avoid it's nearest neighbour. Within a short space of time, it will have then approached another neighbouring Boid, which will react to avoid it. The movement of this one will then consequently affect the surrounding Boids, resulting in a wave of movement from the centre of the flock. In other words, every movement within the flock will be reflected by a movement of the flock as a whole. This would actually well describe one of the aspects of a realistic flock.

My own algorithm also chooses to use the 'area of influence' model in a slightly different way. Each Boid will only act on one of the rules of behaviour at one time, depending on its distance from the nearest neighbouring Boid. If the Boid is too far away, it will approach the neighbour; if it is too close, it will move to avoid it. At a distance between these boundaries, where the Boid need no longer move away or towards it's neighbour, it aligns itself to the neighbouring Boid.

In this way, we can see that the model works by having a series of boundaries at which the rules are implemented. This is where I believe my own algorithm may be optimised by the application of a genetic learning algorithm. There are several parameters which may be adjusted in my algorithm, and possibly more which I will consider in the next chapter, which could be optimised to produce the best flock. For example, altering the avoidance distance parameter, the Boid speed, or the turning speed, will all contribute to an overall change in the appearance of the flock.

5. Proposed Extension To The Boids Algorithm

As a learning exercise and a method of research, I want to apply the knowledge that I have gained of Machine Learning and Genetic Algorithms to implement a learning mechanism to a Behavioural Animation system which will optimise the performance of a flock. Behavioural systems are reputably difficult to program such that they are aesthetically equivalent and convincing with regard to the observations we can make in nature, and they are also deceptive in that despite apparent technical perfection, they can still fail to match the simplicity afforded by natural processes.

This is why I am hoping that, by applying some of the lessons learnt from natural processes such as evolution and natural selection, I can seek to optimise the behaviour of a flock. As I have found in my research, Genetic Algorithms are highly robust, and as a result should be capable of seeking out the optimal parameters for the system. Whilst they also are capable of being entirely automated (i.e. let the program run until it comes up with its own optima), I think it is also possible to work in an element of human interaction to the program so that the user can pick out the aesthetically pleasing hypotheses and generate subsequent populations from those. I will discuss this issue in greater detail later in the chapter.

As a basis for my implementation, I will be using a program that I have written before (see Appendix A) which is a flocking system based loosely on the Boids Algorithm by Craig Reynolds. Whilst this program does not demonstrate perfect flocking characteristics, the basic rules regarding flock behaviour are implemented. I noticed as well that the program makes use of various key elements that I can see will be useful when implementing the Genetic Algorithm; parameters which the algorithm can seek to optimise.

In this chapter I will be describing the method that I would use to go about building the behavioural system so that it may be used with a GA to optimise it, speculating on theoretical design choices to assist in this implementation, and referring to my own and the research of others to guide my choices. It is also likely that my current version of the behavioural system will need to be added to in content to provide the framework on which to base the learning algorithm.

5.1 Outline Of System

The learning system that I propose to use is based on the fact that the parameters used to construct the flocking system can be altered by the user manually in order to find a better behaviour. My idea is to use a genetic learning algorithm in order to use the computer to find the optimum behaviour more quickly than might be possible by trial and error.

I have established that the typical parameters on which the target function might be based are as follows:

- Number of Boid collisions
- Variation in average heading (tests alignment)
- Average speed
- Number changes of direction
- Speed of direction change
- Distance between Boids

Some of the above can be changed within the coding of the flock itself, whilst others are qualities which arise as a result. For example, it is not known before the flock simulation is run, how many collisions there will be, but we can program such parameters as the distance at which Boids will turn to avoid each other, thus driving the distance between them. Similarly, the Boids speed can be controlled, as can the distance travelled before a change in direction is affected.

The outline of the program therefore will be to firstly produce many instances (hypotheses) of the flock with different values for the parameters above. This will create the initial population (first generation) of hypotheses. Each hypothesis must then be assigned a 'fitness value' using some manner of target function. A fraction of this population will then be selected by means of probabilistic selection, and a further fraction be modified by genetic operators based on their 'fitness'.

5.2 Machine Learning Applied

My review of the techniques involved with Machine Learning have taught me that it will be useful in this problem to break up the task into smaller sections. Whilst it is difficult to define the behavioural system accurately as described in Chapter 2, it is a useful exercise in simplification. Therefore, I have established the following about this problem:

- Task *T*: To create a realistic, naturally behaving flock simulation.

The real task here is define the behavioural characteristics of individual flock members such that the overall behaviour of the flock is a convincing simulation of the real thing. The search space for such a task is so large that I have concluded that the use of Genetic Algorithms would be a good idea, since they provide a robust, thorough search technique.

- Performance Measure *P*: The behaviour of the flock; similarity to real flocks/schools.

I think that the only way one can really ensure that the learning algorithm tends towards an aesthetically convincing flock is to compare it to a real one, as observed in nature. I will discuss the methods that I intend to achieve this later in this chapter. However, I feel that any mathematically or logically defined target function which attempts to define the movement of a flock will be flawed, and as such, new ways must be sought out to find a way round this problem.

- Training Experience *E*: The training experience is driven by the genetic algorithm; it will be the production of new hypothesis populations by the implementation of genetic operators.

The training experience is literally the programs creation of each successive population; by creating new populations it will tend towards a specific result. This is, in effect, its training experience.

5.2.1 Target Function

My research into the area of machine learning has suggested that the way that programs may learn is to define a target function which will provide them with the ability to assign fitness to various hypothesis. However, for my purposes, I think that I need to come up with a slightly different solution for assigning fitness.

The problem with the flock is that it is driven by various different parameters which make up it's behaviours, and indeed, its overall appearance. Whilst it may be possible to create a target function based on these parameters and general observations of the flock behaviour, I don't feel that this would guarantee the genetic algorithm finding an aesthetically pleasing behaviour. A target function based on parameters would only work

if one knew the precise behaviour of a flock in nature; however this is not currently the case. Many people have worked on simulating behavioural systems, only to end up defining an algorithm which *approximates* flocking behaviour. The Boids algorithm for instance, makes general observations about facets of flocking behaviour which can be recreated with programming techniques, and the simulation may even be very close. This may be because nature cannot be recreated with mathematics, or that it would be simply too complex to find the mathematical expression which describes the behaviour.

My theory is, in order to expect a computer program to learn what is the best solution to simulating a behaviour, we must give it the best example; nature itself. This may be compared to a painter being asked to paint a picture from direct observation, or from a photograph of the scene. With direct observation, the painter can make his own interpretation of the scene, and approximate it to create his painting. In the case of using a photograph however, the painter is already working from someone else's approximation. The photographer has chosen the field of view and focal length, whilst the camera has captured only a representation of the colour, light conditions and atmosphere of the scene. The painter is then expected to recreate the scene accurately using limited knowledge of the original scene. It is clear which of these approaches will yield the better painting of the original scene.

5.2.2 Zaera, Cliff and Bruten – User Interface Target Function

During my research I came across another research paper in which Genetic Algorithms had been attempted to be used to evolve a flocking system. Zaera, Cliff and Bruten correctly identify that the manual design of flocking system controllers is a difficult task and could perhaps be better implemented using a learning system:

> "Most of the work with animats has involved manually designed controllers, and the indications are that the design of such controllers is a difficult task. Artificial evolution would seem to offer a route by which much of this hard work can be avoided."
> (*Zaera N. et al, (Not) Evolving Collective Behaviours In Synthetic Fish, '96, p1*)

The findings of their research however was that whilst it was relatively easy, and they were successful in recreating simple behaviours such as aggregation and dispersal, the evolving of flocking behaviours was not successful.

One of the main reasons that they propose for their failure to evolve flocking behaviours was that the target function was inadequate for the purpose of assigning a fitness value to the flock hypotheses. They argue that a valid target function would require not only optimising to multiple optima (dependent on the time the simulation has been running), but also requires optimising multiple objectives. (Michalewicz '94, cited by Zaera et al, '96).

They then continue to argue the case that in the absence of a sufficiently complex evaluation function, the only way to ensure the evolution of a convincing behaviour would be for human interaction to be used to monitor the process.

This was indeed the conclusion that I had come to as well when considering this problem. The logical choice would be to get the user to decide on the fitness of various flocks, and then to allow the Genetic Algorithm to process the results of this selection process. In this manner, the program is only half-automated, but it reduces the complexity and time required in programming in favour of a natural selection process.

Since the end use of this evolution of a flock is to be used for an aesthetics purpose after all, does it not make sense to involve the user in the process of selection, such that he/she has an influence on the resulting flock? The most important and over-riding factor in this simulation is that the flock convinces the human eye that it is real, therefore, I feel it should be a human eye which decides how it looks in the evolution process. In this way, the user can guide the evolution, and work towards a result which is convincing, rather than the GA simply working to satisfy a collection of rules defined by a target function.

However, in order to better prove my decision, I also looked into another method of fitness assignment, which I will explain in a later chapter.

5.2.3 User Interactive Fitness Selection

The method required for the user to interactively select a fitness value requires a visual review of the current population. This technique has been used extensively by various researchers to demonstrate Genetic Algorithms. The field of Evolutionary Art, pioneered by William Latham often uses methods of user-interaction with evolution in order to produce art which the user has a preference towards. This is another reason that I came to the conclusion that this would be a viable solution for the flocking system.

In Evolutionary Art, the user interactively selects those images that are preferable, and then the genetic algorithm works on the user's selections in order to generate the subsequent population. This process can be repeated as many times as is necessary until the desired effect is achieved.

Some examples that I have researched can be found at: www.netlink.co.uk/~snaffle/form/descriptions.html#biomorph_commercial

These programs, particularly Form by Andrew Rowbottom, and Imogenes by Harley Davis have led me to a solution to my own problem. These programs feature a user selection as a form of target function. Both work by selecting one parent hypothesis, from which the succeeding population is generated. I feel though that this slightly limits the operation of the Genetic Algorithm in that we only have one parent hypothesis. A much more suitable method would be to allow the selection of a group of hypotheses. It also came to my attention that the 'fitness value' in these programs must only be 1, or 0,

depending on whether the image was selected or not. This does not compare to the approach suggested earlier whereby the fitness value would be a floating point value between 1 and 0, indicating the level of fitness for probabilistic selection for the next population.

I decided to see if I could find any other methods which could be implemented for user selection which would allow the user to assign a more accurate fitness value to individual hypotheses. My research led to the work of Anargyros Sarafopoulos, who has implemented systems which allow the selection of hypotheses using a Graphical User Interface similar to that of programs like Form. Examples of his work on creating evolved textures using Genetic Algorithms may be found at: http://ncca.bournemouth.ac.uk/main/staff/anargyros/gp_images/gp_images.htm

These images however involved a user-interface which allowed the user to give the hypotheses (images) individual ratings. In the book "Handbook Of Computer Animation" (John Vince, 2003), Sarafopoulos' chapter on "Evolutionary Algorithms In Modelling And Animation" describes the Genetic Algorithm technique and also depicts the User Interface used for his Evolutionary program. On this basis, the user is assigning fitness values for the hypotheses, and thus the operation of the Genetic Algorithm works on those. It can be observed that, using this method, the need for a defined target function is negated, since the user defines his/her own target as their personal preference. Thus, an element of control may be maintained by the user in terms of how they would want the result to look, or in my case, behave.

5.3 Genetic Algorithm Pseudocode

Here is the outline for the final algorithm that I would like to write in order to apply Genetic Algorithmic learning to Behavioural Animation systems. I will present the solution in terms of steps, and attempt to explain my reasoning for the decisions.

The areas which I have designed are:

- The Behavioural System
- Definition Of Parameters
- Encoding Of Parameters Into Binary
- Genetic Algorithm Technique and Pseudocode
- Graphical User Interface (Theory)

5.3.1 The Behavioural System

For the remainder of the design of this algorithm I will assume that I have a fully working Boids algorithm implemented in OpenGL. In actual fact, I have an approximation to the Boids algorithm which I coded prior to this research paper. I have included the source code for this in Appendix A for reference.

The main missing parts from my version are some minor areas of functionality, including collision avoidance at the edge of the tank, where due to a coding error the fish will pass through each other instead of steering to avoid collision. Another problem is the lack of a change in speed of the fish, although I have included a variable which allows the speed to be changed.

Besides these, other variables would need to be added in order to provide the parameters on which the genetic algorithm will work, which I will discuss in the next section. Other than this, we will make the assumption that the Boids algorithm has been correctly implemented.

5.3.2 Definition Of Parameters

My research has suggested that the main driving forces with regard to behavioural systems regard the behaviours of *separation*, *alignment*, and *cohesion*. I propose that these variables take place at different specified distances from individual Boids, following the assumption that the Boid will make the decision to carry out a specific behaviour depending on how close it is to neighbouring Boids. I believe that the genetic algorithm will be able to determine what the optimal settings are for these parameters such that the correct behaviour is witnessed.

Other variables which may affect the behaviour of the flock include the average speed of the Boids, and their variation in speed, since if they are highly varied then the flock will

not stay together well. A cohesive flock would surely demonstrate similar speeds for every Boid? This is something for the genetic algorithm to discover.

Rate of change of angle (i.e turning speed) must also be observed and altered to see its affect on the flock. Also, number of turns per minute may be measured and altered. My original algorithm does not implement this behaviour, since it forces the Boids to turn only at the edge of a specified area.

Therefore I have specified the following parameters to drive the flock behaviour:

- Distance at which Boids will separate
- Distance at which Boids will align
- Distance at which Boids will move towards each other.
- Speed (a maximum speed)
- Speed (a range value which determines how much speed can change)
- Rate of change of heading (a value specifying degree by which a Boid can turn per frame)
- Number of direction changes per minute

Next, I need to define a range for these parameters, in effect closing the search area so that the genetic algorithm knows where to look for the best solution. I want to determine where the solution is likely to be, giving enough room to cover all eventualities but also not too large a search space. This is difficult to consider without an example, which is why I have included my source code in Appendix A. In this example, the Boids are set in a 3-dimensional space, determined by x, y and z co-ordinates. As a starting point, my typical example defines the SIGHTDIST (the distance at which the Boid will swim towards another Boid) as 100. Similarly, ALIGNDIST is 20, and AVOIDDIST is 10.

These values may differ depending on the implementation of the system, but I will use these for the sake of example. Some typical ranges therefore would be:

Parameter name	Minimum	Maximum	Range
SIGHTDIST	10	500	400
ALIGNDIST	5	450	445
AVOIDDIST	1	100	99
SPEED	0.01	0.5	0.49
SPEEDRANGE	0	0.5	0.5
TURNDEGREE	0.0001	0.05	0.0499
NUMTURNS	0	100	100

The next stage is to consider how these values may be encoded into binary for the operation of the Genetic Algorithm.

5.3.3 Encoding Parameters To Binary

The 'range' column in the table above gives an idea of how to encode the above parameters into binary. We must have enough bits to cover these ranges with the appropriate level of accuracy.

One way of making this process easier is to force the search range to be equivalent to a nearby binary border, such that each change in the bit string is equivalent to a one unit change in the parameter. For example, the new ranges would be:

Parameter name	Minimum	Maximum	Range	Change per bit
SIGHTDIST	10	522	512	1
ALIGNDIST	5	517	512	1
AVOIDDIST	1	127	128	1
SPEED	0.01	0.65	0.64	.01
SPEEDRANGE	0	0.64	0.64	.01
TURNDEGREE	0.0001	0.0511	0.0512	.0001
NUMTURNS	0	128	128	1

In this way, a one bit increment will be equivalent to a one unit change in the real parameter range.

With the above coding method, the parameters would need the following number of bits to represent them:

Parameter Name	Number Of Bits Required
SIGHTDIST	9
ALIGNDIST	9
AVOIDDIST	7
SPEED	6
SPEEDRANGE	6
TURNDEGREE	9
NUMTURNS	7
Total	53

This will make the total bit string length 53 bits, which makes up the entire genotype for the flock. A simple rule for decoding the genotype allows for the string to be split up again once the genetic alterations have been applied, and then decoded to be used as parameters for the flock.

5.3.4 Pseudocode

This is my proposed design for the program, which comprises of pseudocode language. I intend for this design to be sufficient for someone else, if not myself, to be able to read and understand this method and be able to implement the program from reading this.

Procedure GA (*fitness_user_selection*, *fitness threshold*, *n*, *c*, *m, t*)

Where: *fitness_user_selection*: the function which initiates the GUI for user assignment of fitness values. The function runs a series of video files for the user to assign fitness levels interactively (see section 5.3.5).

fitness_threshold: the level of fitness at which the program will terminate and use the current best hypothesis.

n: number of hypotheses to be generated per population

c: the fraction of the population that is replaced by the crossover operation

m: the mutation rate

t: time in seconds for each flock simulation

- Generate the first population P (n hypotheses generated at random)
  - Run flock for *t* seconds per hypothesis
  - Outputs video file and save
- Run GUI for user selection of fitness levels; fitness_user_selection. Accept a fitness for each hypothesis.
- While [complete = 0] do

  Create a new generation, P_s:
  - Select $(1 - c)n$ hypotheses from P to add to P_s. The probability $Pr(h_i)$ of selection of any particular hypothesis h_i from P is given by:

    $$Pr(h_i) = \text{fitness\_user\_selection}(h_i) / \Sigma^{n}_{j=1} \text{fitness\_user\_selection}(h_j)$$

  - Apply crossover operation to ½(n.c) pairs of hypotheses from P according to above expression. For each pair $\{h_1 , h_2 \}$, apply the crossover operator to produce two child offspring. Add offspring to P_s.
  - Choose m percent of the population P_s at random, with uniform probability. For each, invert one randomly selected bit in its bit string.
  - Update P_s to be the new current population, P.
  - Run GUI for user selection of fitness levels again; fitness_user_selection.
- Return the hypothesis from P which has the highest fitness level.

The first step of the program will be to generate the first population at random, according to the set of rules defined in the previous chapter regarding the ranges of the parameters. These are then used to generate the flocks for each given hypothesis. Once the program has chosen a hypothesis, the flock is instantiated, and the result is captured as video output and saved temporarily. These files are volatile, and will be replaced by the subsequent population to save memory. Alternatively, the user may be able to choose an

option which keeps the generations in a separate cache, such that all generations may be viewed.

The Graphical User Interface is then run in which the user may assign the fitness for each given hypothesis. Using these fitness values, the program will then apply the genetic operations in order to generate the subsequent population.

It should be noted that the genetic operations can be altered by changing the variables of *c* and *m*. The user interface itself would have options to change these variables, since these describe the manner in which the next population will be created. For example, a smaller value of *c* will cause the subsequent population to be less varied from the previous population, since less of the hypotheses will be selected for genetic crossover. This will likely mean that more generations must be produced before an adequate solution is reached, but this may be a useful approach when the results are tending towards finality.

Similarly, the mutation rate *m* may be altered, where again the user would observe that subsequent populations differ from current populations by a smaller margin. Mutation however is designed to alter hypotheses by incrementally small amounts, and therefore should not affect the population change as drastically as the crossover operation.

The program is designed only to stop creating new populations when the user selects a particular flock as 'Complete'. This would be a ticker box which denotes that this is the hypothesis to be used, and as such the program need not generate any further hypotheses.

5.3.5 Graphical User Interface

The user interface that is required should allow the user:

- To review flock simulations in real-time
- To see all flocks simultaneously and also separately
- To select a sliding value for each hypothesis denoting their 'fitness'
- To change mutation and crossover rate
- To change the 'simulation time' *t* in order to better review flocks
- To change the number of 'animats' in the flock
- To allow selection of a final hypothesis
- A Testing Interface, allowing selection of one hypothesis to test it with different numbers of animats and larger simulation times.

In order to meet these criteria, the design that I have in mind combines many of the features that I have seen used by existing programs aforementioned in my research (Form, Andrew Rowbottom; Imogenes, Harley Davis), as well as a few of my own ideas specific to solving these problems.

The interface would consist of a full screen view of the current population; a set of thumbnail video clips showing the hypotheses (flocks) being played back together for time *t*, and looping them. The user can then deduce from looking at this screen which of the flocks deserve the highest merit. It may be the case that these thumbnails have to be 2-dimensional captures above the flock, a top-down view which conveys the necessary information but at a lower performance cost.

The user can then select individual flocks to view by clicking on them, bringing up a larger view of the simulation. This is a new interface which should allow the user to rigorously test the currently selected hypothesis. The user would have the option to request a re-simulation of that individual flock using the current hypothesis, thus displaying a real-time, 3-dimensional view of the flock. Simulation time *t* and the number of animats within the flock may also be changed. This will allow the user to make the correct decision when fine-tuning his/her decision on fitness level.

The user must assign fitness on the thumbnail screen, in which sliders by each of the thumbnails can be moved to select the fitness level. (The sliders give a floating point value to the program which denote the hypothesis fitness). Sliders may be left at 0, ensuring that these hypotheses will not be selected for the following population. This is useful if, for instance, the program has produced an anomalous hypothesis which results in very inadequate behaviours!

This screen also has a checkbox next to each thumbnail, which should be selected if the user wishes to use this flock as the final solution.

There would also be various drop-down menus which allow the user to alter the crossover and mutation values for generating the next population, such that different settings can be tested to see which yield the best results.

Finally, for performance control, the user should also be able to request the program to pause the population generation at any time, in case performance is unacceptably low. The user may also choose to change the individual flock simulation time *t*, to provide quicker feedback on the new population.

6. Project Evaluation

I began the research for this project with the intention of finally implementing the idea to use Evolutionary Techniques to develop Behavioural Systems. Unfortunately, for a variety of reasons, I was unable to accomplish this goal. I did however, manage to research the subject to a level of understanding that, should I want to in the future, I should be able to go ahead and implement the ideas. I have found the field of Genetic Programming and Evolutionary Techniques to be compelling and exciting; I only regret that I have not had the time to implement the things that I have learnt.

I began with early research into the work of Craig Reynolds, since for a while I had thought about investigating Behavioural Systems themselves for the Innovations project. Having already implemented my own flocking system, I knew a little about them prior to this research, and wanted to know what the current lines of innovative study in the area were. It was in this manner that I came across the idea of Machine Learning, and subsequently, Genetic Algorithms. I then took some time to research these areas.

Due to another idea that was presented to me, I changed my mind with the project and researched an innovative way in which I could model the University campus, hoping to contribute towards my Major Project. I was introduced to an idea that 3-d models of building-sized objects could be produced by 'scanning' them with a device recently acquired by the department. However, soon enough it became apparent to me that not only were there no tutors who could really help with this kind of project; I also realised that it was a highly technical project, since the data received from the scanner would have to be analysed, somehow converted to a language understood by the 3-d package, and then constructed in such a way that multiple shots of a model can be 'stitched' together to make a complete model.

I deduced that, regretfully the project was not for me. At this late stage was where I got back to the original idea, and hit upon the plan to create a user-guided evolutionary algorithm to evolve flocking systems. My research gave me the background knowledge to come to the conclusion that this method should work.

I feel that, although I have learnt so much about this field, there is so much left to be learnt. I have barely scratched the surface of the plethora of related fields to Genetic Algorithms. Neural Networks, for example, fall almost hand-in-hand with Genetic and Evolutionary Programming, but I have not touched on them in this paper. I have, however, covered the elementary understanding of this field of research and feel rewarded for undertaken this research report.

Appendix A

Boids Algorithm Example Code

Acknowledgements

Thanks to Ari Sarafopoulos for agreeing to tutor me on the project at a late stage, and thanks for your help and advice.

Bibliography

Bäck, T., Fogel, D. and Michalewicz, T. (eds.) (2000) Interactive Evolution. *Evolutionary Computation 1: Basic Algorithms And Operators*, p. 212 – 225. Institute Of Physics Publishing.

Callan. R. (2003) Genetic Algorithms. *Artificial Intelligence*, p. 330 – 345 Palgrave-Macmillan.

Goldberg, D.E. (1989) *Genetic Algorithms In Search, Optimisation & Machine Learning*, p. 1 – 88. [n.p.]: Addison-Wesley.

Holland, J.H. (1975) *Adaptation in natural and artificial systems*. Ann Arbor: The University Of Michigan Press.

Langdon, W.B. and Poli, R. (2002) Fitness Landscapes. *Foundations Of Genetic Programming*. p. 17 – 26. Berlin: Springer-Verlag.

Man, K., Tang K. and Kwang S. (1999) *Genetic Algorithms*, p. 1 – 62. Berlin: Springer-Verlag.

Mitchell, T.M. (1997) Genetic Algorithms. *Machine Learning*, p. 249 – 270. Singapore: McGraw-Hill.

Reynolds, C. W. (1987) Flocks, Herds, and Schools: A Distributed Behavioral Model, in Computer Graphics, 21(4) *SIGGRAPH '87 Conference Proceedings,* p. 25-34.

Rowbottom, A. (1997) *Descriptions of Programs which produce evolutionary art* [online] Available from: www.netlink.co.uk/~snaffle/form/descriptions.html#biomorph_commercial [Accessed 28 February, 2005]

Sarafopoulos, A. [n.d.] *Images Generated Using Interactive Selection and Genetic Programming* [online] Bournemouth: Bournemouth University. Available from: http://ncca.bournemouth.ac.uk/main/staff/anargyros/gp_images/gp_images.htm [Accessed 4 March, 2005]

Vince J. (2003) Evolutionary Algorithms In Modelling And Animation. In Sarafopoulos, A. *Handbook Of Computer Animation*, Berlin: Springer-Verlag.

Zaera, N., Cliff, D. and Bruten J. (1996) *(Not) Evolving Collective Behaviours in Synthetic Fish*. Published in *From Animals to Animats 4* (SAB96).