An Introduction to: Evolutionary Computation and Genetic Algorithms

Christopher Stephen Tingley
MEng. Computer Science and Cybernetics

29th March 2005

Abstract

Evolutionary computation is a broad name for several incarnations of evolutionary algorithms (EAs). In general, EAs maintain a population of individuals which evolve based on a number of stochastic and algorithmic techniques applying to the selection, recombination and mutation of each individual within the (often randomly) created population. Several different types of EA will be explored, their similarities and differences illustrated, before implementations of two EAs are described and the influence of varying techniques and parameters on their results are discussed.

Contents

1 Introduction **5**

1.1 Why use evolutionary algorithms? 5

1.2 Applications for evolutionary algorithms 5

2 Background **6**

2.1 Genetic Algorithms (GAs) 6

2.1.1 Representation 6

2.1.2 The Evaluation Function 7

2.1.3 The Fitness Function 7

2.1.4 Selection 8

2.1.5 Reproduction 8

2.2 Evolutionary Programming (EP) 9

2.3 Evolutionary Strategies (ES) 9

3 Methods **10**

3.1 Task 1 10

3.2 Task 2 11

3.3 Task 3 12

3.3.1 What is the Prisoners' Dilemma? 12

3.3.2 What is the Iterated Prisoners' Dilemma (IPD)? 12

3.3.3 Evolutionary Techniques and the IPD 12

4 Results **14**

4.1 Task 1 14

4.2 Task 2 14

4.3 Task 3 17

5 Discussion **17**

5.1 The Evaluation and Fitness Functions 17

5.2 Parameter Tuning 17

5.3 And Finally... 18

6 Conclusions **18**

7 Apendix **19**

7.1 Matlab code to produce function graph 19
7.2 Matlab code to overlay points on function graph 19
7.3 C# Code for Generic Genetic Algorithm 20
7.4 C# Code for Binary Encoded Chromosome and Population 24
7.5 C# Code for Roulette Wheel Selection 28
7.6 C# Code for Fitness and Evaluation Functions 29
7.7 Screen Shots 30

8 References 31

List of Figures

1 The structure of an evolutionary algorithm 6
2 Binary representation of the knapsack problem 7
3 Sexual reproduction illustrating single and multiple point crossover . . 8
4 The function to solve for tasks one and two 11
5 Single peak genetic algorithm: results (population size: 10) 15
6 Single peak genetic algorithm: results (population size: 25) 15
7 Single peak genetic algorithm: best evaluation over time 16
8 Multi-peak genetic algorithm: results (population size: 10) 16
9 GUI for Task 1 30
10 GUI for IPD 30

"In the broadest sense, evolution is merely change, and so is all-pervasive; galaxies, languages, and political systems all evolve. Biological evolution ... is change in the properties of populations of organisms that transcend the lifetime of a single individual. The ontogeny of an individual is not considered evolution; individual organisms do not evolve. The changes in populations that are considered evolutionary are those that are inheritable via the genetic material from one generation to the next." [1]

1 Introduction

Genetic algorithms and the doctrine of evolutionary computing is a rapidly growing field of research. Whilst based upon characteristics we see within biology, the similarities are idealised and are there; simply to provide a convenient analogy. In the way we can see evolution within nature, we can picture evolution within evolutionary computation. The basis behind this being simple: a population or possible set of solutions is evaluated, with the best solutions being used to breed a new set of solutions. In the biological sense, very much like Darwinism [2].

The term evolutionary computation actually refers to several distinct areas, namely: genetic algorithms, evolution strategies, evolutionary programming and genetic programming. At this stage it is important to note that, evolutionary algorithms (EAs) are stochastic (random) in nature and as a result do not guarantee the ability to produce the optimal solution. They do however, confess to having the ability to produce optimum solutions and as a result of their stochastic nature, any one EA may produce a different optimum solution on a consecutive run. For this reason, it is common for EAs to be run many times allowing better solutions to be found and for the best overall solution to be used.

1.1 Why use evolutionary algorithms?

Put simply, the power of evolution is extraordinary. Breeding a dominant selection of individuals from any one population may subsequently create a better population. Do this several hundred, or maybe even thousands of times and it is possible to create a population of solutions which contain solutions far superior to any that were first started with.

By applying this ability to create a superior population with a complex real-world problem, an evolutionary algorithm can provide solutions to the problem in ways that would otherwise be difficult for a traditional algorithm. The reason for this is due to that the evolutionary algorithm (and programmer) need not have an in-depth mathematical understanding of the problem to be solved. This makes evolutionary algorithms very powerful as they can often be implemented in a very short space of time and often on very complex problems.

1.2 Applications for evolutionary algorithms

Evolutionary computation is primarily an optimisation technique that can be applied to a wide range of problems. Further into this paper, it will be demonstrated that a genetic algorithm can provide optimum solutions to a problem where it would be tedious to exhaustively explore the solution space (especially to find a solution to a large number of decimal places). Furthermore, it may be impossible for a mathematical representation of a problem to be created, therefore rendering a mathematical solution inaccessible. In this case, an evolutionary method can be applied to retrieve a solution that was otherwise deemed impossible - assuming of course, that a representation suitable for an evolutionary method can be devised. Different representations will be considered later into this paper.

2 Background

As introduced above, an EA is an iterative process where a population is continuously evolved until an acceptable solution is found. There are several different ways to create the new population but, these are explored further on in this paper. The process is depicted graphically in Figure 1.

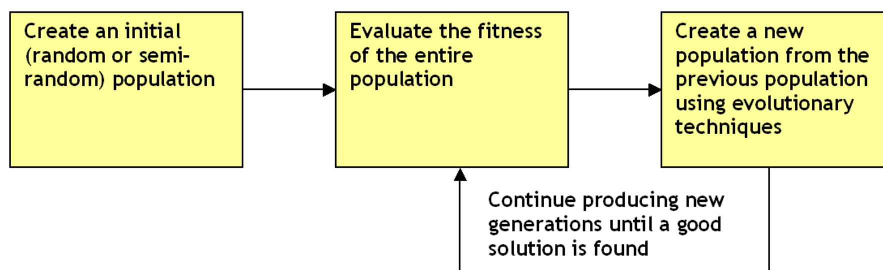

Figure 1: The structure of an evolutionary algorithm

The process behind creating a new population, based upon the current population is known as a generation. Forming a new generation comprises of three stages: selection, reproduction and replacement. There are several techniques to each of these stages and these are explored within the context of genetic algorithms in the following section.

2.1 Genetic Algorithms (GAs)

The GA paradigm is analogous to biology in several ways, most apparently in their use of the terms such as chromosome, allele, phenotype and genotype, reproduction and generations.

A GAs' population is constructed of chromosomes (also known as individuals or members). Each chromosome encodes a solution, where the encoding (or genetic composition) is known as the genotype and what it represents is known as a phenotype. As a result, it follows that the same genotype can represent many different phenotypes based on how the chromosome is evaluated. It also follows (and possibly easier to visualise) that a phenotype can be encoded as different genotypes. An example here may illustrate this point a little more clearly, take for instance, a date. The date first of April 2005 in the UK is abbreviated to 01/04/05, however in the USA the same abbreviation is written 04/01/05. This is an example of the same phenotype (the date) being represented as two distinctly different genotypes.

2.1.1 Representation

The genotype may be represented in a number of ways. The most common (and easiest to implement) is binary encoding, where a string of ones and zeros represent a distinct solution. This can be illustrated using the knapsack problem where there are several items to be placed into a knapsack. Each item has an associated size and there is a

limited amount of room within the knapsack. Each item is assigned a index within a binary string and if the item is to be placed in the knapsack, it is represented by a one, if not, a zero. See Figure 2.

Compass	Map	Water Bottle	Penknife	Machete	Flare
1	1	0	0	1	0

Figure 2: Binary representation of the knapsack problem

Representation could also be of a real numeric value. For example, 6 binary digits could represent the decimal numbers between zero and 64, or could just as equally represent 64 equal fractions between zero and one. The possibilities of representation are endless and are only limited by the solution domain in hand.

2.1.2 The Evaluation Function

Chromosomes must be evaluated in a way appropriate to their representation. For example, if a genotype is constructed of binary digits, it may represent integers in binary code or grey code. When designing evaluation functions it is trivial to evaluate a individual and obtain a solution, but particularly more difficult to find a solution to the problem being attempted. For this reason, it is important to always have the objective in mind when constructing evaluation functions, as nobody wants an answer to a question they haven't asked!

Another important factor involved with evaluation functions is the time taken to evaluate the solution. If a GA is being used to design a game playing strategy for example, in order to evaluate a proposed solution, it may be necessary to run the strategy through a simulation which may take, say, 15 minutes. If you have a population size of 40 and require 1000 generations to produce an adequate solution, the time taken to simply evaluate the solutions would be over a year! For this reason, it is important that the evaluation function is correct - as mistakes could be very costly.

2.1.3 The Fitness Function

After evaluating each chromosome, it must be passed through the fitness function. This function is very much dependent on the purpose of the GA and is the principle method of deciding which chromosomes within the population are good and which are bad. When maximising a value, the fitness function is fairly trivial and will simply rank each chromosome dependent on how far off the current best evaluation it is.

If finding a series of 'good' solutions, the fitness function becomes slightly more complicated as it must find the same fitness for varying evaluations. Using the problem explained in 4.2 as an example, there are four peaks contained within the solution domain, all of which must be found - not just the best. This requires a rework of the fitness function in order for smaller peaks to be given the same fitness as larger peaks.

Parent chromosomes

Parent A	1	0	0	1	1	1	0	0	1	1	0	1
Parent B	0	0	1	1	0	0	0	1	0	1	1	0

Single point crossover

Child A	1	0	0	1	1	1	0	1	0	1	1	0
Child B	0	0	1	1	0	0	0	0	1	1	0	1

Multiple point crossover

Child A	1	0	0	1	0	1	0	0	0	1	1	0
Child B	0	0	1	1	1	0	0	1	1	1	0	1

Figure 3: Sexual reproduction illustrating single and multiple point crossover

2.1.4 Selection

Elitism, or "survival of the fittest" is a very powerful selection process, used within a lot of GAs. As the name suggests, a certain percentage (usually around 20%) of the best solutions (the elite) are taken from the current generation and are placed directly into the new generation. This ensures that good solutions are never lost between consecutive generations and of course, are likely to be good chromosomes to breed with. Similar to elitism, truncation selection will only breed the best of the population in order to create the new population.

Tournament selection, possibly the best of the selection techniques randomly picks two (or more, sometimes seven) chromosomes from the population and picks the best to go through to reproduce. This process does give a slim chance to pick a bad solution, but in the main will always pick the stronger chromosomes. Another good selection method is roulette wheel. Although not as good as tournament selection, roulette wheel also gives poor chromosomes a chance to be picked for reproduction, this does have its downside in that it can cause the GA to take longer to converge on a solution. Ranked selection is very similar to roulette with trivial differences.

2.1.5 Reproduction

After selection has taken place, there are two main ways of producing new chromosomes, namely sexual and asexual reproduction. It is usual for GAs to combine both methods to ensure strong children are bred (maintaining strong alleles) and that a small amount of random variation occurs within the population. This random variation is akin to the genetic defects which occur in nature and avoid breeding converging to just one gene pool.

Crossover (or sexual reproduction) is where two chromosomes are used to create two new chromosomes (children), the principle is easily illustrated with binary represented chromosomes. With single point crossover, a random point within the chromosome is selected and the child chromosome is created from the binary digits before the crossover point from the first parent and the binary digits after the crossover point from the other parent. For multiple point crossover, the same process is used, but with several crossover points, see Figure 3.

Variations on crossover include the 'headless chicken crossover' which combines a selected chromosome with a randomly created chromosome and 'one-child crossover' which creates two child chromosomes as with standard crossover, except one child (at random or otherwise) is discarded.

Mutation (or asexual reproduction) does not use parent chromosomes to produce children, but simply makes a copy of itself along with a slight modification in order to create a new chromosome. Mutation is usually used in conjunction with crossover and is often used sparingly ($\approx 0.01\%$ of the time). The purpose behind mutation is to jump solutions out of local optimums in an attempt to move towards the optimal solution, but has varying effects based on the representation chosen. For example, when using simple binary coding, if bit two within a four bit genotype (representing the integers between 0 and 15) is changed from a zero to a one, this will affect the evaluation by four, effectively altering the solution by 26%. This is obviously a very large change and can be avoided by using codings such as grey code.

2.2 Evolutionary Programming (EP)

The principles behind evolutionary programming (EP) are much the same as GAs and due to their similarities become easier to illustrate after a understanding of GAs has been ascertained. As with GAs, EP assumes that solutions can be represented as a number of variables and a fitness value for each solution can be calculated. EP is different from GAs in two major ways, the first being that there is no constraint on the size of the representation. In terms of binary encoding, this could mean that extra bits can be added and removed from the representation. Often, the representation within EP closely matches the problem domain, for example in real-valued optimisation problems, the individuals within the population represent real-valued vectors [3].

Child production is where the second major difference between EP and GAs lies. Unlike reproduction in GAs, EP traditionally only uses mutation operations on the parents in order to produce children, crossover (recombination) is rarely used. The reason for this is that the fundamental principles behind EP relies on recombination between species (individuals). It is often the case that a Gaussian distribution of mutation operators is used, ensuring that large mutations happen infrequently and more subtle mutations happen frequently. As mutation operations have differing severity it is also standard practice to use less severe mutations when nearer the optimal solution. This approach provides a number of benefits, the most important being that for optimisation problems, an EP will converge faster on a solution than an GA. Individuals within a EP population are selected using the same techniques as used in GAs. It is more important to produce more individuals in a new population and to only keep the best individuals of the population, as it is difficult to predict the outcome of a (often scholastically) mutated chromosome.

2.3 Evolutionary Strategies (ES)

Evolutionary strategies (ES) share many of the same characteristics as EP and GAs. An important difference between ES and GAs (as with EP) is found with the genotype of the individuals. Operations on the genotype are usually direct and not via some sort of encoding, unlike GAs which can encode their genotype using differing phenotypes.

As with EP, Gaussian mutations are applied to each parent within the population, but then a selection method is used to determine which new members of the population are to be removed (i.e. killed from the population). Again, similar to EP, self-adaptive methods are used to select the most appropriate mutation to use at each stage of evolution. In contrast to EP, ES use deterministic selection in order to decide which of the members of the population to kill, based upon the evaluation of each member.

3 Methods

Three tasks were set out to explore in more detail some of the aspects of evolutionary computation as introduced above. It was suggested that task one and two were approached using a GA, whereas free reign was given over the third task. A generic evolutionary algorithm class using a binary representation was written within the custom `BinaryEncoded` namespace. The source code is available on the accompanying CD and in appendix 7.3 through 7.6. It will be shown that slight changes to the GA class written for the first two tasks, bring the GA class and algorithms more inline with an EP approach.

3.1 Task 1

The first task consisted of maximising y for the equation shown in 1. This is also shown graphically in Figure 4. As can be seen by the figure, there are four peaks within the solution domain, one of which is very near an extremity and another two which are very close in evaluation. There are also several pitfalls in the form of local minimums.

$$y = x + 8sin(4x) + 6cos(5x) : 0 \leq x \leq 2\pi \tag{1}$$

The language chosen for the implementation was C#.NET and as a result a generic, binary represented (for any real number) GA/EP class was created (c.f. 7.3) which can be used within any .NET program simply by including the `BinaryEncoded` namespace and creating an instance of `GeneticAlgorithm`. Once a `GA` object has been created (using a constructor giving total control over the genetic algorithm parameters), two methods must be assigned to the object, these are: `GA.EvaluationFunction` and `GA.FitnessFunction`. As the class created is completely generic, the evaluation and fitness function can be specified by the user. A series of other methods including visualisation of the individuals and of the entire the population are also available from within the class, giving a powerful insight into the workings of the algorithm.

As described above, a basic binary representation was used, with tests using several different sizes of representation, differing selection techniques and a multitude of various parameters. The number of bits in the representation ranged from 10 to (the maximum possible using `BinaryEncoded.Chromosome` - see appendix 7.4) 32. The fitness value for each individual in the population was simply the evaluation of the function, as the value of x is being maximised, the better the evaluation and therefore the better the fitness. Elitism, crossover (with tournament selection) and mutation on the crossover produced children were also implemented. The fitness and evaluation functions used for task one can be seen appendix 7.6.

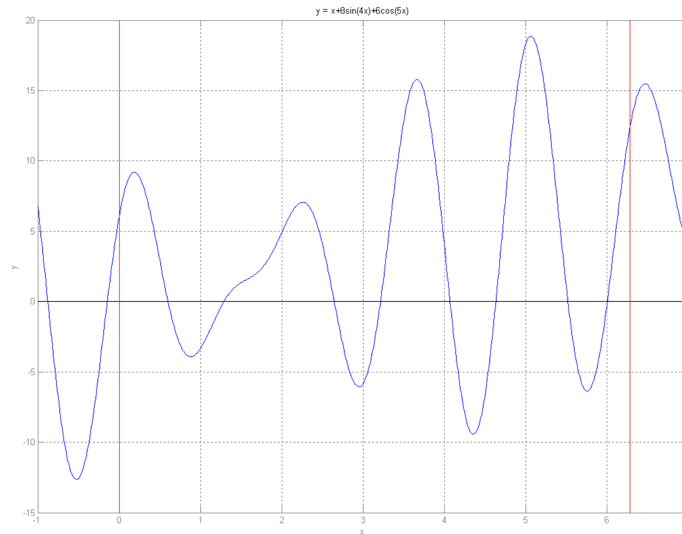

Figure 4: The function to solve for tasks one and two

3.2 Task 2

A few improvements were made to the generic GA class, but by creating a new fitness function, the second task was completed with relative ease. The only difficulty was in determining this new fitness function.

The fitness function was required to produce a high fitness around the peaks of the function. The easiest way to do this was by taking the differential of the function at the point and using that as the basis for the fitness. To take into account that when the differential may equal zero, the peak may be a local minimum, or even a flat part of the function (see Figure 4 where $x \approx 1.5$) a differential over a small neighbourhood was used. This neighbourhooding function is shown below in 2 where α defines the neighbourhood, typically 0.02 - 0.05 and $evaly$ calculates the value of y at any given x value (i.e. the evaluation function).

$$\frac{dy}{dx} = \frac{|evaly(x + \alpha) - evaly(x - \alpha)|}{2 \cdot \alpha} \tag{2}$$

To make the implementation slightly easier, the reciprocal of the neighbourhood function was used and if the evaluation of the chromosome was found to be negative the result from the neighbourhood function was also made negative (see 7.6 for the full implementation). This new fitness function was found to work well, but it was still difficult to stop the GA from converging to only one or two of the best solutions (i.e. solutions with shallower peaks). This was combated by tweaking α in the neighbourhood function and by removing very good solutions (i.e. solutions scoring a very high fitness) from the population. It was at this point that the generic GA class began to look more like a GP than a GA, primarily due to this new variable population size.

3.3 Task 3

For the third task, there were three problem to select from: Connect 4 [5], Iterated Prisoners' Dilemma (IPD) [6] and the Microsoft Imagine Cup 2005 Visual Gaming category (VG) [7]. The task picked was IPD.

3.3.1 What is the Prisoners' Dilemma?

Bill and Ben are both arrested for some heinous crime and are placed in separate isolation cells, no communication between the two detainees is permitted. Both Bill and Ben care more about their personal freedom over the welfare of their accomplice. Once the case comes to trial, the prosecutor lays down a deal to both Bill and Ben. "You may either choose to confess or to remain silent. If you confess and your accomplice remains silent, all charges against you will be dropped. Similarly, if you remain silent and your accomplice confesses, you will do the time and your accomplice walks free. If you both confess, you will both go to prison, but I will ensure that you both receive early parole. Finally, if you both remain silent, it will be harder for us to get a serious conviction, but you *will* both go to jail to serve token sentences."

Neither Bill or Ben know what each other will say and herein lies the prisoners' dilemma (PD) faced by the prisoners. No matter what the other prisoner does, it is better off to confess rather than to remain silent, but the outcome if both prisoners confess is much worse than the sentence they would have received if they had both remained silent.

3.3.2 What is the Iterated Prisoners' Dilemma (IPD)?

A serious of different scenarios can be modeled by the PD including military rivals or price setting for duopolistic firms [8] and in these cases, it is better to model these scenarios by using an iterated version of the PD known as the IPD. Throughout the duration of the IPD, access to the results of previous turns (of both parties) is available. This creates a new dimension to the game where opponents who remain silent (defect) in preceding rounds can then be "punished" by subsequent defections and cooperation can be "rewarded" by further cooperation. Once this is the case, strategy for a single player becomes less obvious and the strategy to adopt becomes dependent on the apparent strategy of the opponent. It is important to note at this stage that it is assumed the IPD is being played against a rational opponent who is actually employing a strategy and is not completely random.

3.3.3 Evolutionary Techniques and the IPD

The IPD since its conception has been a problem which has used evolutionary techniques to come up with new strategies to play the game. The highest scoring strategy in Axelrod's initial tournament was named Tit for Tat (TFT) [6] which simply imitates its opponents previous move. What was interesting is even though the results of the first tournament were made publicly available, the TFT strategy went on to win the second tournament which had sixty three entrants!

		Opponent	
		Defect	Confess
Player	Defect	-1	5
	Confess	0	3

Table 1: Payoff matrix for the IPD

In order to simplify the problem and in the process create a scoring system suitable for use within an EA, a payoff matrix is used (see Table 1). This matrix simply provides both players with a score for each of the four possible outcomes of each game. This can be then used by an EA which is given the task of maximising any one players' accumulated payoff over a finite number of games. The payoff matrix shown above is used by the GP implementation of the third task.

A common approach to generating strategies for the IPD using evolutionary techniques is to encode for n number of previous games within a chromosome, where each chromosome in the population determines a distinct strategy. For example, if three previous games and the move to make in next consecutive game is used, the chromosome would contain $(3^2 + 1) \times 4^3 = 448$ bits, enough to encode every possible outcome for 3 games and the next move. These chromosomes (or strategies) can then be evolved, thus creating new strategies. The strategies can then be played against each other over a series of games, using the payoff matrix to assign a score (i.e. fitness) to each strategy. Once a strong strategy has been developed, the associated chromosome can then be used as a lookup table based on the previous history of the current game and thus providing the player with the next best move.

This approach has its merits, however it seems that once a strategy has been developed and chosen, no matter how many games this strategy may play for - it will always follow its set of pre-determined "best moves" based upon the last few games. If the opponent is able to determine this strategy, it is simple to take advantages of weaknesses within in it. The complexity of determining the strategy being used by the opponent is obviously intrinsically linked to the number of previous moves the strategy is based upon. This means that if a strategy based on a large amount of previous moves is used, the more unlikely (and exponentially longer it would take) for the strategy to be determined. The problems associated with increasing the number of previous moves are simple to illustrate, for example imagine a chromosome encoding all possible outcomes for a game history of only 10 moves - this would equal a chromosome string size of 6,464! Evolving a large population of individuals this size would take a considerable amount of time and may present a problem when finding the game history within this string at run time.

For the reasons stated above, it was decided that an "on-line evolution" approach would be better, as it would adapt to the opponent, making it impossible for the opponent to determine the strategy being used. This approach would also allow the constantly evolved strategy to take advantage of any generosity exhibited by the opponent. The chromosome encoding as explained above was used however, the difference being that the evolution was only run on the previous two or three games ensuring that evolution times were not too large - as the number of possible outcomes for two or three games is fairly manageable. And of course, evolution (with random strategies) was run after each game to develop the strategy for the next move.

4 Results

4.1 Task 1

Due to the small range of x, it would of course, be possible to exhaustively search the problem domain. With a binary representation of 24 bits, the number of solutions is $2^{24} = 16,777,216$. Obviously with modern computing, this may seem to be trivial, however this is only a small problem and as either the number of dimensions, or the number of bits increase, this number increases exponentially. The resolution can be calculated by dividing the number of solutions we can represent by the solution space domain, in this case (with a 24 bit representation): $\frac{2\pi}{16,777,216} = 3.7451 \times 10^{-7}$. This provided an sufficiently high resolution for this problem. Of course, if a higher resolution was required, the flexible nature of the GA class made it trivial to change the number of bits used by the representation.

The GA was seen to perform exceedingly well, even a relatively small population size. A population size of only (≈ 10) would be able to quickly converge on a good solution, but would require more generations than if a larger population size was used. This was beneficial as it meant that convergence on a solution took several generations and allowed a better look at what was happening within the population.

Figures 5 and 6 show the evaluation of each member of the population plotted on top of the function being solved at 0, 10, 20 and 50 generations. It is clear that in both cases, even after 20 generations a significant percentage of the population has converged on the highest peak within the solution space. Figure 7 shows the evaluation of the best chromosome within the population at each generation on a typical run of the GA. This shows that after as few as 10 generations (with a population size of 10) a very good solution is found. This solution is not really improved significantly for another 30 generations where upon the solution is not seen to improve anymore.

4.2 Task 2

A much larger population size was required for the second task. This was needed in order to provide a suitable covering of the solution space when the initial population is created. Once a large coverage was provided, the algorithm was able to converge on all of the solutions in the domain easily and with more reproducibility.

As with many evolutionary techniques, a small amount of domain knowledge is often needed - in this case it was known that the local minimums were all negative evaluations, as a result the fitness function was easily improved by putting this domain knowledge into practice. This is obviously not an ideal solution, but it worked very successfully and therefore provided an adequate fitness function for the task in hand.

The selection method used for both this and the other tasks was tournament selection with variable number of competitors varying from two to seven. Lower numbers tended to breed more diverse children, and was useful with this task. For task one however, a larger number of competitors worked better, as only the best solution was required and genetic diversity was not as important. Roulette selection was tried, but tended not to yield particularly good results in terms of the time taken to converge on solutions.

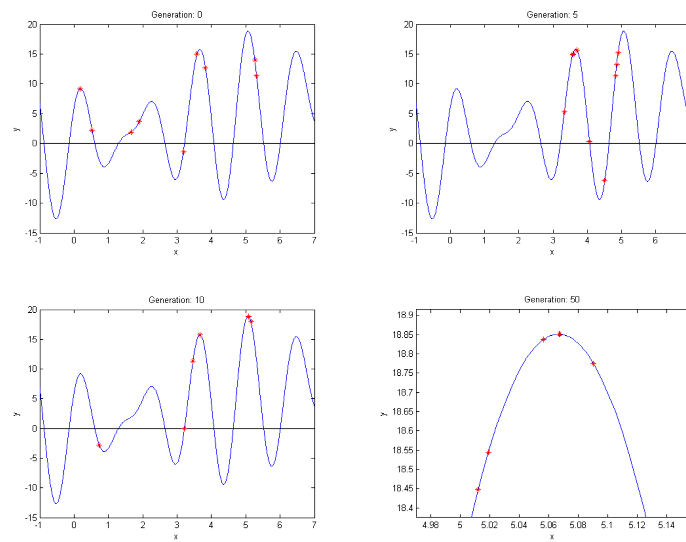

Figure 5: Results of the genetic algorithm finding single-peak (population size 10) after 0, 10, 20 and 50 generations. Elitism 20%, Crossover 80%, Mutation 0.05%.

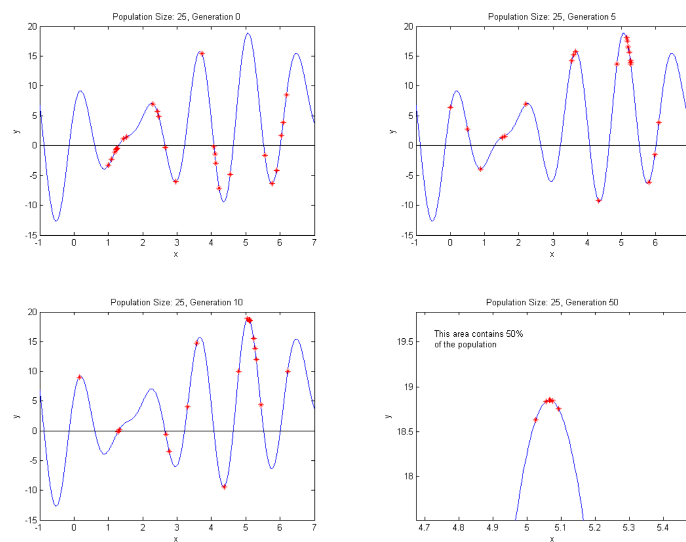

Figure 6: Results of the genetic algorithm finding single-peak (population size 25) after 0, 10, 20 and 50 generations. Elitism 20%, Crossover 80%, Mutation 0.05%.

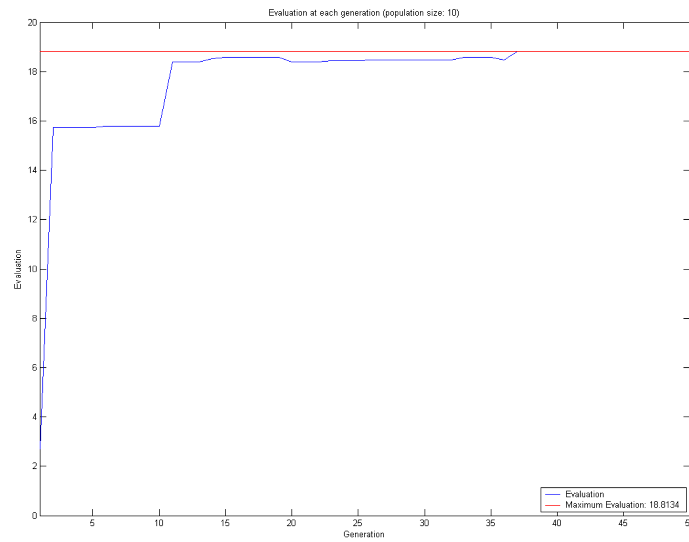

Figure 7: A plot showing the evaluation of the best chromosome within the population at each generation. The red line indicates the best evaluation reached, as can be seen this is achieved after approximately 40 generations

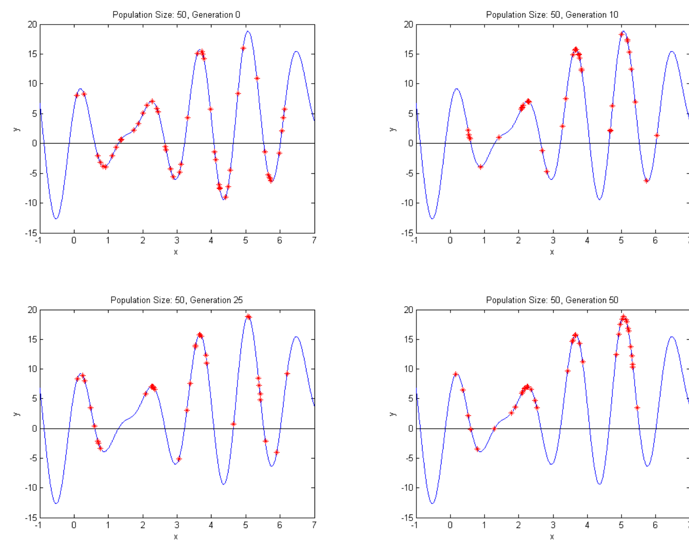

Figure 8: Results of the genetic algorithm finding multi-peaks (population size 50) after 0, 10, 20 and 50 generations. Elitism 20%, Crossover 80%, Mutation 0.05%.

4.3 Task 3

It was difficult to draw many results from the IPD as the implementation was never completely finished. For this reason, a code listing is not included in this report as it is the author's intension to finish the implementation however, a copy of the source code is available on the accompanying CD. Preliminary results are promising as it can be seen that the GP can adapt to changes in the game play of the opponent.

5 Discussion

5.1 The Evaluation and Fitness Functions

The crux of the methods implemented within this project are obviously contained within the evaluation and fitness functions. If these are wrong, there is no chance of success, at least not obtaining any sensical (or even wanted) results. If they are correct however, there is every chance of succeeding. The better these functions are - the better the results and more often than not the quicker the results are produced.

Herein of course, lies the difficulty with evolutionary algorithms. Domain knowledge has been touched upon earlier in this paper, but it is prudent that it be brought up again at this stage. Without it; it is significantly difficult to create both the evaluation and fitness function, but if a great deal of domain knowledge is known, it begs the question: can this problem be solved without going to the expensive of developing an evolutionary approach?

5.2 Parameter Tuning

Population size is possibly the most interesting variable to alter within EAs. Too small and it is unlikely that an adequate solution will be found, too large and the computation time is horrendous. It is obviously more important to get a decent solution, so a large population is recommended in order to provide a suitably large (and hopefully random) coverage of the solution domain. Tasks one and two are limited to one dimension, however the problem of solution domain coverage is likely to be much more prominent with multi-dimensional problem domains. Another possibility with continuous variable optimisation problems is to use a pattern of initial samples giving an even coverage of the search space. This can avoid the problem of randomly creating a group of initial individuals in bad areas, be it a local minimum or possibly worse, a local maximum.

Setting other parameters such as mutation rate, crossover rate and choosing different selection methods causes the output of the algorithms to change drastically. For this reason it is important that different combinations of these parameters are tested if a successful implementation is to be achieved. Without this experimentation it is exceedingly difficult to predict how an EA will perform on a problem, unless there has been some previous experience of deciding upon parameters in a similar situation. Even then, the EA may behave differently than expected. The trouble here is: how are the parameters tested? Especially as they may interact in complex ways.

5.3 And Finally...

The advantages of evolutionary techniques are clear, predominantly the ability to produce solutions to problems were the solution domain is too vast to explore explicitly. Due to this staggering ability, the overall impression of these techniques is good, but again; it is clear that a certain amount of knowledge regarding the problem is needed - especially when it comes to setting the parameters of the algorithm being employed.

Proof of results, especially with optimisation problems is troublesome. Due to the low reproducibility of results (even more so with large solution domains), it is difficult to prove that a solution given by any on EA is the best. All that can be proved is that it may be better than another solution, but this is due to the intrinsic stochastic nature of these algorithms and should be taken into account when considering using an EA.

6 Conclusions

The methods discussed in this paper are only touched upon lightly, the field of evolutionary computation is massive with a great deal of research taking place within each of the areas introduced. After the generic `BinaryEncoded.GeneticAlgorithm` and each of its associated classes and methods was completed, a firm grasp and an underlying comprehension of the fundamental (and in some respects, more advanced) areas of evolutionary computation was obtained.

Encouraging results were produced by the algorithms, but the inherent problems of an evolutionary approach were all too apparent, particularly the importance of domain knowledge. A long list of improvements and other techniques could have been included within the `BinaryEncoded` namespace, however it will no doubt be useful as a very good starting block to any other work that the author embarks upon within the realms of evolutionary computation.

The techniques discussed in this paper are certainly interesting and there is obviously a lot of scope for more research within this field. The number of problems that can be tackled using an evolutionary approach are astounding due to the flexibility of the algorithms in their design of the representation, evaluation and fitness functions. Designing the right tool for the right problem is the principle difficulty and for this reason, whether they will ever be found in that toolbox kept by modern day programmers - well, that remains to be seen.

Acknowledgments: The author would like to thank the following people for their help: Dr. William Browne (supervisor), Dr. Ben Hutt and Mr. Philip Crabtree (fellow undergraduate). The resources and articles in [9] to [20] are also hereby acknowledged.

7 Apendix

7.1 Matlab code to produce function graph

```
 function cs3m2
% Plots:  y = x+8sin(4x)+6cos(5x)
x = (-1:  .01:  7);
y = x + 8*sin(4*x) + 6*cos(5*x);
plot(x,y)
hold on
plot((-1:0.005:7),0, 'k')
plot(0, (-15:0.005:20), 'r')
plot(2*pi, (-15:0.005:20), 'r')
title('y = x+8sin(4x)+6cos(5x)')
ytitle('y')
xtitle('x')
grid on
```

7.2 Matlab code to overlay points on function graph

```
 function plotresults(results)
% Plot a 2 variable (x, y) array
hold on
for ct = 1:length(results)
    x = results(ct, 1);
    y = results(ct, 2);
    plot(x, y, 'r*')
end
```

7.3 C# Code for Generic Genetic Algorithm

7.4 C# Code for Binary Encoded Chromosome and Population

7.5 C# Code for Roulette Wheel Selection

7.6 C# Code for Fitness and Evaluation Functions

7.7 Screen Shots

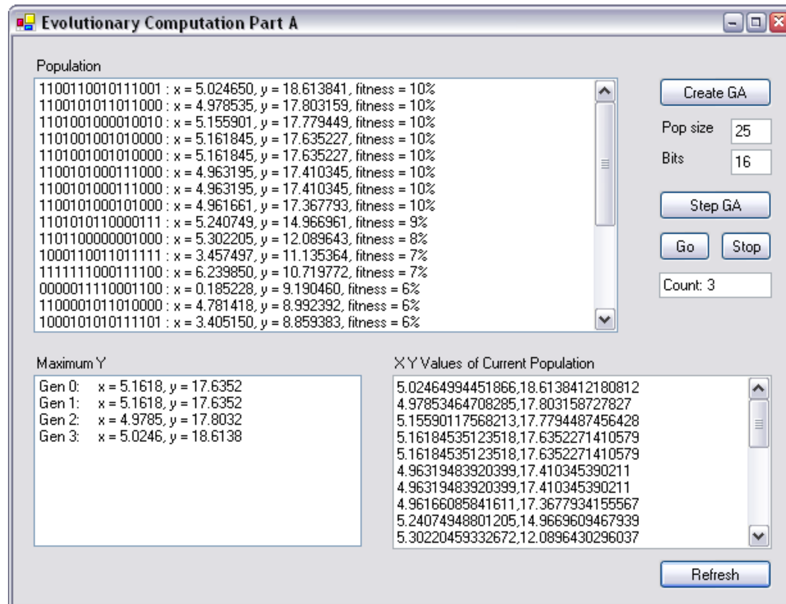

Figure 9: GUI for Task 1

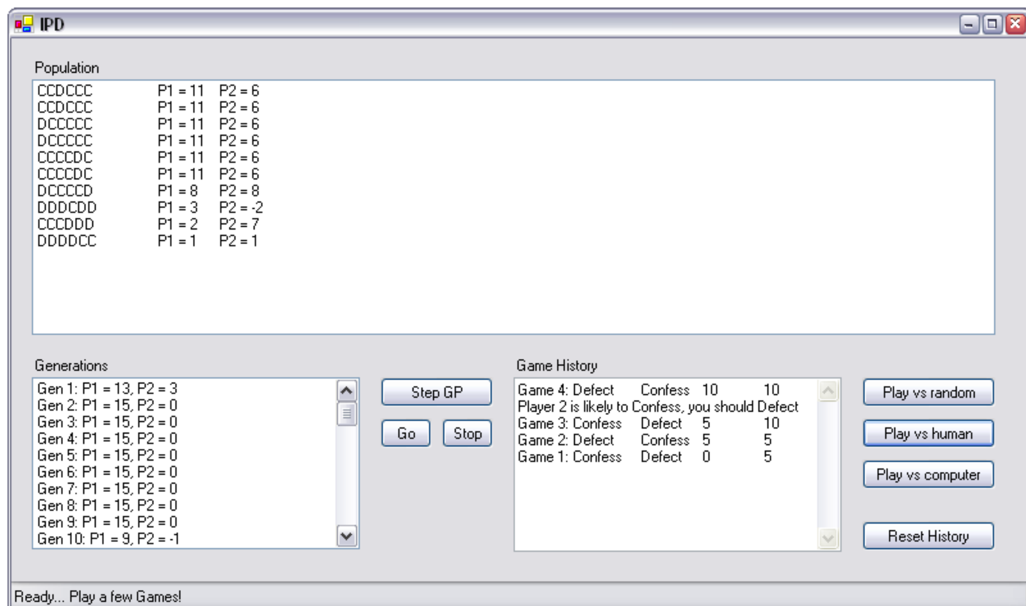

Figure 10: GUI for IPD

8 References

[1] D. J. Futuyma, *Evolutionary Biology*, Sinauer Associates, 1986

[2] *Definition: Dawinism*, (2005), [Online] Available: http://en.wikipedia.org/wiki/Darwinism

[3] W. Spears, K. De Jong, T. Baeck, D. Fogel, H. de Garis, *An Overview of Evoluationary Computation*, 1993, European Conference on Machine Learning. Available: http://cs.uwyo.edut/ spears/overview/ecml93.all.html

[4] Z. Michalewicz, D. Fogel, *How To Solve It: Modern Heuristics*, Springer, 2002, pp. 161-183, pp. 277-301.

[5] *Connect IV Research Site*, (2004), [Online] Available: http://sip189a.reading.ac.uk/

[6] R. Axelrod, "The Evolution of Strategies in the Iterated Prisoner's Dilemma", in L. Davis (ed.), *Genetic Algorithms and Simulated Annealing*, London: Pitman and Los Altos, CA: Morgan Kaufman, 1987, pp. 32-41.

[7] *Microsoft Imagine Cup 2005, Visual Gaming Category*, (2005), [Online] Available: http://thespoke.net/visualgaming

[8] S. Kuhn, "Prisoner's Dilemma", in E. N. Zalta (ed.), *The Stanford Encyclopedia of Philosophy*, 2003, Available: http://plato.stanford.edu/archives/fall2003/entries/prisoner-dilemma/

[9] G. F. Plum, *Genetic Algorithms*, (1998, Jun.), [Online] Available: http://www.uni-koblenz.de/ kgt/Learn/Textbook/node122.html

[10] M. Obitko, *GA Recommendations*, Czech Technical University, (1998, Sept.), [Online] Available: http://cs.felk.cvut.cz/ xobitko/ga/recom.html

[11] C. Ferreira, *Gene Expression Programming: Mathematical Modeling by an Artificial Intelligence*, 2002, Available: http://www.gene-expression-programming.com/gep/GepBook/Introduction.htm

[12] J. Nye, *Genetic Algorithms: Explanation and Implementation Tradeoffs* (2000, May) [Online]. Available: http://www.devmaster.net/articles/genetic-algorithms2/

[13] *Evonet*, [Online] Available: http://evonet.lri.fr/

[14] B. Keller *Notes on Evolutionary Algorithms*, [Online] Available: http://www.cs.hmc.edu/claremont/keller/152-slides/

[15] *The Hitch-hiker's Guide to Evolutionary Computation*, [Online] Available: ftp://ftp.cerias.purdue.edu/pub/doc/EC/Welcome.html

Produced using LaTeX