CS3M2 - Evolutionary Computation

Philip Crabtree
Department of Cybernetics
University of Reading

The field of evolutionary computation exists to develop methods and techniques which can be implemented in to any given problem in order to find the most optimum solution through the mutation and breeding of a set of possible solutions. This breeding and mutation is often based on Darwin's theories and are designed to mimic real world evolution They can prove to be most useful when the designer of the system does not actually know the solution. This paper will focus mainly on the branch of evolutionary computation known as Genetic Algorithms and their uses will be demonstrated through two function optimisation tasks and the techniques will finally be adapted to produce a solution for a Connect-4 playing system.

1. Background

These are many available options and design strategies available to one who wishes to create software which aims to solve a given problem. The area this paper is concerned with is that known as Evolutionary Computation. More specifically the concern lies in the field of research called Genetic Algorithms which will be the main focus although other branches of evolutionary computation will be discussed.

A Genetic Algorithm aims to simulate the evolution of life from a poor population to a rich population. The basic concepts behind evolutionary techniques are found when looking at Darwin's work on the evolution of life. Genetic algorithms generally start with a poor population although it is possible that, by chance, the optimum solution for the given problem is found when initialising the population. The stages of a simple genetic algorithm are as follows. (note that full explanations of each step are given later in the paper)

1. The population (of a given size) is created by randomly creating a gene structure.
2. The populations fitness is assessed
3. Members of the population are selected and bread to form offspring for the new generation
4. Some members of the new population are mutated.
5. Steps 2 – 4 are repeated until a maximum number of generations has been met.

1.2 Representation

Possible solutions for the given problem need to be represented in a form that lends itself to easy manipulation that to some extent mimics real world evolution. The most common method of representation is a binary string, sometimes represented as grey code. This is ideal as real values can be found by simply decoding the binary string and scaling as necessary as well as allowing for easy breeding between two members of the population. The exact techniques used for breeding, or crossover as it is commonly known, are discussed later.

For the problem given as the basis of this report the encoding is relatively simple and is just a binary string which is decoded to give a solution

011010011101

However, the representation also lends itself to more complex representations whereby one string can represent two or more parts of the solution (e.g. an x, y and z value);

0110 1001 0101 1000 1

Binary strings are not the only way to represent a possible solution. Other methods may be to use real values although this leads to complex algorithms when performing crossover operations etc and are generally unnecessary as a binary string can, when decoded, represent a real value.

1.3 Fitness

This is perhaps the most important phase of the genetic algorithm as it determines what future generations aim towards.

In the case of the given problem being to find the maximum solution of a given function, the fitness can simply be the result of the function. The higher the result of the function, the higher the fitness and hence all the members of the population will evolve towards that higher value. More complex fitness functions are normally necessary and may often require extra functions such as fitness sharing (discussed later). A more complex fitness function may look at previous histories of the population and their results to determine if its current fitness assessment is correct. This is almost like having a secondary genetic algorithm within the primary genetic algorithm.

1.4 Elitism

The concept of elitism can best be summed up by the well known phrase “survival of the fittest”.

A population is given an elitism value which represents how many members, often expressed as a percentage, of the current generation are to be kept, unchanged, for the next generation. This is done to ensure that at least some of the best current solutions are carried forwards rather than running the risk of mutating all the members to a low fitness.

1.5 Selection

Many forms of selection have been suggested for genetic algorithm applications. Amongst the most popular are the tournament and roulette wheel methods. Each selection method has its advantages and disadvantages and are more suited to a particular type of problem than others might be. Below are two of the most common selection techniques although many more do exist.

1.5.1 Roulette Wheel Selection

As the name suggests, this technique can be represented as a roulette wheel. The wheel is divided up into a number sections which totals the sum value of all the fitness’s of all the members of the current generation. In this way, a member who has the fitness of one hundred is allotted one hundred slots (out of a possible, for example, one thousand). A member who’s fitness is only equal to twenty is allotted twenty slots. Hence, when the roulette wheel is spun (which is simulated by generating a random number between one and the sum total of all fitness’s) the member with a larger fitness has a higher chance of being selected.

This process is done twice in order to find two possible parents. These parents are then bred to form two offspring.

1.5.2 Tournament Selection

There are many methods of tournament selection. The one most used is one which selects, at random, four potential parents. The four potentials are then put in a tournament to find the best two which are then put forward for mating.

Variations on this method includes the crowding technique which, when an offspring is created, the current population members are analysed and the member most genetically close (i.e. the one who's binary representation differs the least from the offspring) is replaced. DeJong put forward the idea that a number of members from the population are selected, all of which are close (or as close as possible) to the offspring created. The number selected depends on a crowding factor (CF). These individuals are then compared and the worst of these is replaced by the offspring. This is known as the worst of the most similar technique.

1.6 Crossover

When two members of the population are mated to form offspring they go through a cross over procedure. The most common procedure is to have one cross over point although for some applications, more points are required. The actual crossover procedure and results can best be explained by diagrams. Below it can be seen haw two parent are split up at a given point and the genes behind that point are swapped over between parents.

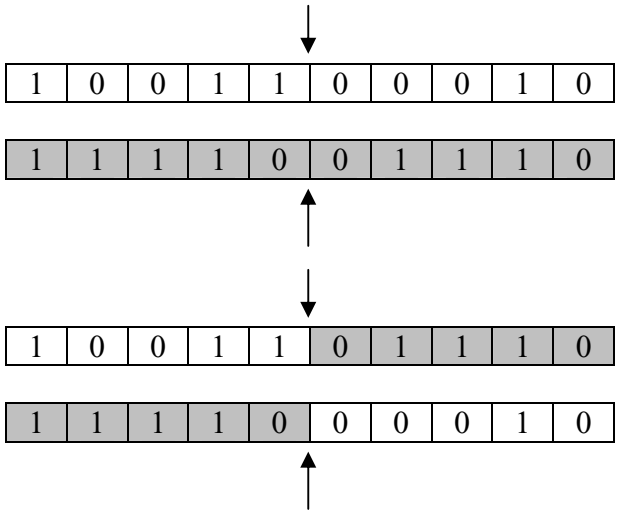

Figure 1.1 Crossover

1.7 Mutation

Mutation is implemented to prevent the population falling into a local minima or maxima. When an offspring is created it is subject to mutation. The actual mutation value can be varied but is generally small enough to ensure only very few (if any) of the members are mutated. Mutation will also generally affect only one part of the gene but can easy be expanded to affect more. If, as is normally the case, the solution, and hence to offspring, are represented as binary strings, then the actual effect of mutation is to simply alter one bit to its opposite (i.e. a 0 becomes a 1 and visa-versa).

1.8 Multi Modal Fitness

In many cases, more than one final solution will be required. This starts to cause a problem when looking at how to evaluate the fitness of a particular solution as, in the case of finding several maximum peaks of a function, a peak of less height will have a lower fitness level (if the traditional techniques are used) and hence the members will start to evolve towards the highest fitness, ignoring the other peaks. To combat this several techniques have been suggested. Amongst these are fitness sharing and the crowding methods. The crowding method has been briefly discussed in the previous sections and it essentially ensures that when an offspring is created it will not just be placed into the next generation but instead it will replace its closest matching parent (in terms of genotype). The closest parent is simply found by finding the hamming distance between the offspring and its parents to find the one with the lower hamming distance. Note that for this to work correctly, the binary string must be encoded as grey code. Replacing the closest parent (which may sometimes result in a less fit solution) ensures that the population covers a good spread of the population space rather than crowding towards the one maxima. The actual effects of crowding can be seen in the results of task 2 in section 3.2.

Fitness sharing is a technique implemented at the time of assessing an individuals fitness. Essentially all the members within a given niche have their total fitness summed up and the average is found. This is then divided by the number of members within that niche. This ensures that if too many members crowd round any niche, the fitness of that niche is lowered to such a level that it becomes undesirable and other potential solutions are sought.

1.9 Other Methods than Genetic Algorithms

The other possible evolutionary routines can now be summarised as the basics of all methods remain very similar (perhaps with the exception of LCS) to a genetic algorithm and all its fundamental parts (crossover, fitness assessment etc) remain similar to those described above.

1.9.1 Evolutionary Programming (EP)

Evolutionary programming (EP) uses very similar techniques to those used in genetic algorithms but with some important differences. Perhaps the most obvious difference is that EP does not, in general, use a crossover operator but instead just mutates the parents to form offspring. It is possible to create more than one offspring per parent and the number of offspring need not match the number of members in the current population. Evolutionary programming is normally concerned with writing a program that is capable of writing another program. This is also true of genetic programming discussed later in this section.

1.9.2 Evolutionary Strategies (ES)

Evolutionary Strategies (ES) are typically applied to numerical optimisation and are normally applied to real numbers rather than discrete variables. Each member in the population is represented as two real vectors. One represents a place in the search space consisting of several real-values, whilst the other represents the standard deviation. Again, like EP, generally no crossover operation takes place and the

population is evolved only through mutation. The actual mutation process is one that ensures that smaller changes are far more likely to happen between generations than large changes (as is true with real world evolution). One such mutation process is displayed below [35]:

$$x^{t+1} = x^t + N(0,\sigma)$$

Where $N(0,\sigma)$ is a random, Gaussian number, with a mean of zero and a standard deviation of σ.

1.9.3 Genetic Programming (GP)

Genetic Programming is a similar technique to those used in Genetic Algorithms but instead of trying to produce an optimal solution from a program, a GP will actually create a computer program and try to evolve that to give the best solution. Languages such as LISP have been designed to allow easy adaptation of the code for just this purpose. In these cases, the fitness is not beast on the output of the program but rather on the program itself, normally in terms of efficiency.

1.9.4 Learning Classifier Systems (LCS)

Learning Classifier Systems (LCS) are based on a set of simple logical rules (known as classifiers) each of which follow an, **if <condition> then <action>** structure [8]. They were originally introduced by Holland as a method of applying evolutionary techniques to machine learning problems. This is perhaps best illustrated as below:

Condition	Message	Matched
1##0	1010	Yes
1##0	1011	No

The result is rarely Yes or No but is more often a real value expressed as either a value or an action. By putting a LCS in combination with other evolutionary computing techniques, one can develop a basic system where by the output of the LCS action is evolved. Perhaps when applied to a path finding algorithm as seen in [8], the output can be evolved to decide which, if given the option, is a better direction to turn in or even perhaps alter the condition through evolution. Instead of using a fitness function as is done with other evolutionary computation techniques, a LCS rule utility is decided through reinforcement techniques.

2. Methods

All the tasks implemented follow the same basic algorithm as displayed below:

```
Main_Loop()
  Initialize_Population();
  for (i = 0; i < POPULATIONSIZE; i++)
        Get_Fitness();
        Get_Next_Generation();
        Swap_Next_Gen_to_Current_Gen();
```

To fully explain the algorithms each function in turn needs to be explained.

2. 1 Initialize_Population();

This creates the initial population by taking each member in turn and giving each of them a random gene string. The actual number of bits in each gene can be altered. When the gene string is decoded it is done in such a way that the total can never equal more than 2*PI as is specified in the task instructions.

Note that this function is actually put in the GA class constructor.

```
Initialize_Population()
  for (i = 0; i < POPULATIONSIZE; i++)
       for (j = 0; j < GENESIZE; j++)
          Rnd = Random_1_or_0();
          this->Population[i].Gene[j] = Rnd;
```

2.2 Get_Fitness()

This function firstly calculates the value of 'x' required by the function. As mentioned previously, the value of x must lie between 0 and 2π and is simply the total of the binary string representation of each member.

Now that the value 'x' of the function specified has been found, the total of the function can be found for each potential solution (i.e. each member). This function total is then stored as the fitness of that member.

Note that there is a secondary check in this function to ensure that the x value produced is defiantly valid. For the most part this is unnecessary however it does ensure that the members do not evolve towards an erroneous value.

```
Get_Fitness()
  for (i = 0; i < POPULATIONSIZE; i++)
    x = this->Get_X_Value_of_Member(i);
    if (x >= 0) && (x<= 2*PI)
          this->Population[i].fitness = x + 8*sin(4*x)+6*cos(5*x);
   else
          X = 0;
```

2.3 Get_Next_Generation()

This function will first sort the members according to their fitness. This is done to ensure that when we later need to select our most elite members it is a relatively simple task. It also proves vital when debugging or just printing data to the screen or to a file for analysis.

The next step is to actually extract or most elite members and copy them to an array representing the next generation. The remaining members of the next generation are found by selecting parents, mating them to form offspring which are subject to mutation and putting these offspring into the next generation. There are several methods available to select the parents as discussed previously. The two methods implemented however are the Roulette wheel selection and the Tournament selection techniques. Pseudo code for the two methods are displayed below:

```
Roulette_wheel()
  for (i = elite; i < POPULATIONSIZE; i++)
    totalSize += this->Population[i].fitness;
    Rnd = RandNum_between_0 and totalSize;
    for (i = 0; i < 2; i++)
      for (j = elite; j < POPULATIONSIZE; j++)
            runningTotal+= this->Population[j].fitness;
            if (runningTotal > Rnd)
                  Parent[i] = this->Population[j]
```

```
Tournament()
  for (i = 0; i < 4; i++)
    Rnd = RandNum_Between_elite_and_POPULATIONSIZE();
    PotentialParent[i] = this->Population[Rnd];
    Sort_Potential_Parents();
    Parent[0] = PotentialParent[0];
    Parent[1] = PotentialParent[1];
```

Now that two parents have been found, they need to be mated together to form offspring which have a chance of being mutated. Note that mutation is performed if a random number is produced which is higher than a pre-defined mutation rate. In the given implementation there are several available methods of mutation, one is to choose a random bit to mutate, one is to count a given number from the left or right, one is implemented to allow several bits to be mutated. One the whole, only one bit need ever be mutated. The whole procedure is done in the following fashion:

Note how the most significant bits are the ones that are most likely to be crossed over.

```
Crossover()
  Rnd = RandNum_between_1_and_(GENESIZE-1)
  for (i = 0; i < Rnd; i++)
     Offspring[0].gene[i] = Parent[1].gene[i];
     Offspring[1].gene[i] = Parent[0].gene[i];
```

```
for (i = Rnd; i < GENESIZE; i++)
  Offspring[0].gene[i] = Parent[0].gene[i];
  Offspring[1].gene[i] = Parent[1].gene[i];
Mutate_if_Necessry();
Copy_Offspring_to_Next_Gen();
```

Mutate()

```
for (i = 0; i < 2; i++)
 Rnd = RandNum_between_0_and_GENESIZE
 if (Offspring[i].gene[Rnd] == 0)
   Offspring[i].gene[Rnd] = 1;
 else
   Offspring[i].gene[Rnd] = 0;
```

2.4 Task 2 Methods

Task 2 requires multiple solutions to be found as opposed to the one maximum value as in Task 1. To do this both the fitness sharing techniques and the crowding techniques were implemented onto of the basic algorithm. In addition to this functions were added to find the hamming distance between any two given gene strings (required for the crowding method).

Fitness_Sharing()

```
Sort_by_X();
for (i = 0; i < POPULATIONSIZE; i++)
  if (Current_x_value <= Previous_x_value + threshold)
        runningTotal +=  Current_Fitness;
  else
        End_Niche();
        All_Members_Fitness_in_Niche =
        runningTotal/Num_members_in_Niche/Num_members_in_Nic
        he;
        Begin_New_Niche();
```

Crowding()

```
Perform_Tornament()
For (I = 0; I < 2; i++)
   Ham[i] =MinHammingDist(Offspring[i], parent[i], parent[i+1]);
   ReplaceClosestParent();
```

2.5 Task 3 Methods

Task three is based on a game of connect 4. The skeleton code is provided by Dan Scott, A Masters student at the University of Reading. Note that whilst task one and two are implemented in C++, Task 3 is implemented in Java as this is the code required in order to correctly interface with the Connect 4 client. The skeleton code provided does nothing more than retrieve and analyse the messages from the client and handle the transmission of messages to the client.

Task three is based around a standard game tree although it differs from the normal game trees used in this situation because the actual evaluation of each move is evolved genetically. The evaluation of each move is based upon the current severity of any position (i.e. if three opponent pieces are lined up then it is a severe case as it is if three of the GA's pieces are lined up ready for a win). Future work could look at evolving the value of severity in any given case. At any given point a decision must be made concerning whether it is better to block an opponents more or place a piece in an attempt to win.

The depth to which the game tree can be evaluated is a variable set from the beginning of a function. The tree is evaluated using the MinMax technique whist also implementing Alpha-Beta pruning to ensure that if a move is a winning or loosing move, the tree is evaluated no further and hence cutting down on processing time. The tree is searched in a depth first manner, where all the possibilities of one given move are explored before moving on to the next possible move. The natural way to perform such a routine is through recursion.

The actual Genetic techniques used are all the same as used in previous tasks but modified to represent the Connect 4 game. The representation of the solution is set as one gene string for simplicities sake but it is actually split into two parts, one representing the value used to evaluate a winning move and the other representing the value used to evaluate a blocking move. The fitness function essentially aims to maximise the evaluation of a winning move and also that of a locking move. However, this can not just performed as such but instead a comparison is needed. Failure to implement a comparison would eventually result in the gene string becoming a list of 1's (as this is the possible maximum. A comparison will ensure that the evaluations are moving in the right direction. That is to say the fitness evaluation is altered by looking at whether or not previous generations were, on the whole, a success. If they were a success then it is ok to assume that its current solution is generally right but will probably require improvement (e.g. a blocking move should take priority over a winning move) rather than the cases where it is decided that the fitness is unacceptable and so is altered to reflect this. Perhaps a bettor solution for future improvements would be to allow this evaluation to be altered genetically.

The standard Connect 4 rules exist and as such, each possible move evaluation must be proceeded by finding the severity of the implication of that move (as mentioned previously). This is done in the normal fashion, through looking at the move horizontally, vertically and through a diagonal. Naturally, one of these takes priority. If the vertical checks are made first then it is possible that no checks will be made for the horizontal and diagonal components. Perhaps one future improvement would be to allow for that order of these checks to be evolved.

3. Results

Tests were carried out on all three tasks. For the first two tasks, the produced results were compared against results from the actual function which was found simply by plotting the function for all values of x. The actual function produced is displayed below:

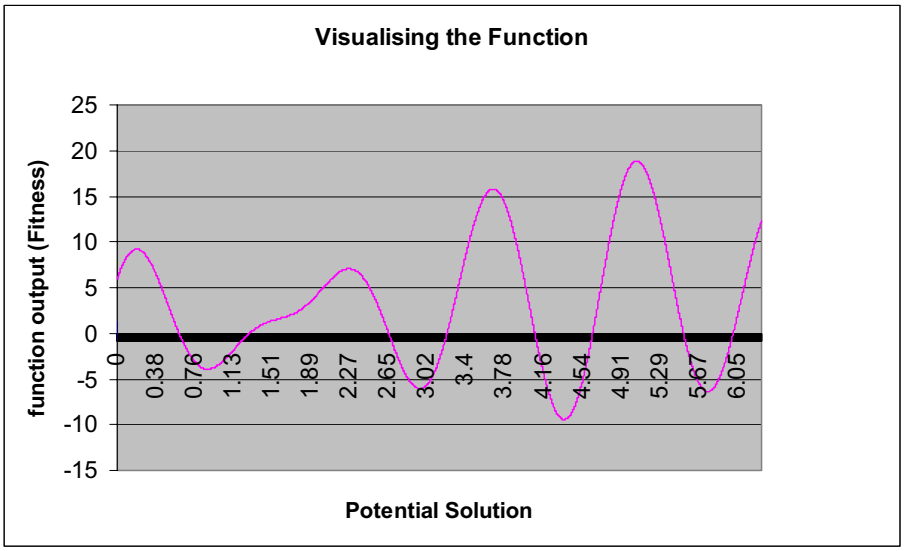

Figure 3.1 The actual function output where 0 >= x <= 2π

As each task has several variables associated with it (population size, mutation rate etc), a normal set of variables was established and each of these was altered in turn to find the effect each variable has on population. Two sets of results were produced, one illustrates the final values of all the members in the population and the other illustrates the 'most fit' member of the population at each iteration.

3.1 Task 1

The normal set of variable used in the testing of this function are:

- 10 Members
- 40 Iterations
- 0.1 Elitism Value
- 0.1 Mutation Rate
- Tournament Selection

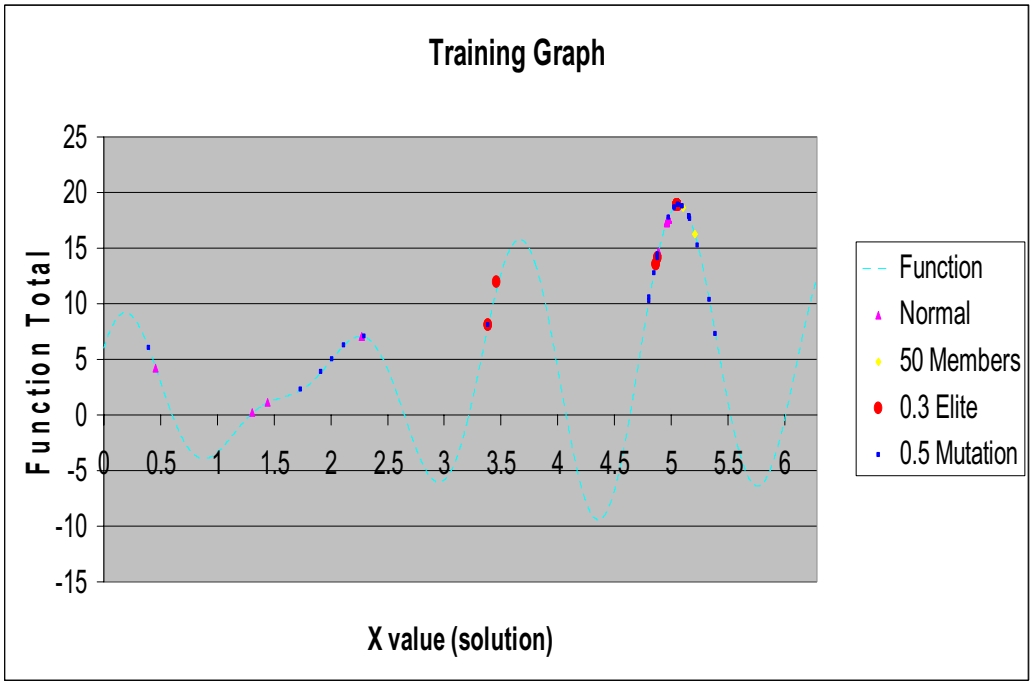

Figure 3.2 Training graph for Task 1.The 'most fit' member at each stage.

Final Solution

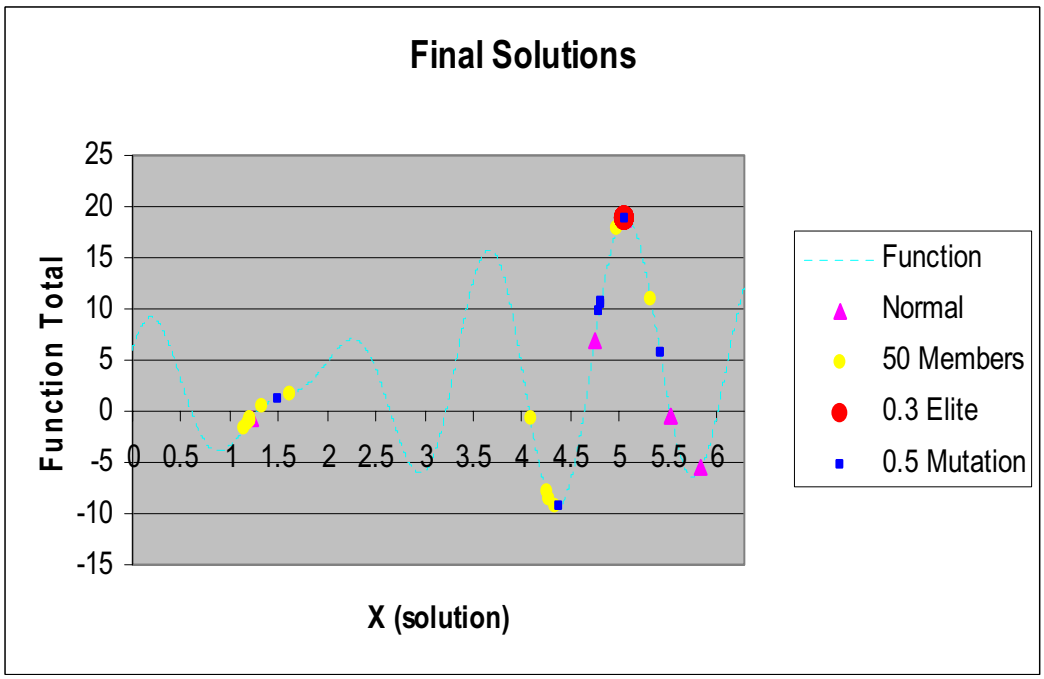

Figure 3.3 Final results for Task 1. The fitness of each member in the final Population.

It is clear from the above figure 3.3 illustrating the final values of each member that the optimum solution is reached in all cases. A particularly good example is the illustration of the population when a value of 0.3 for the elitism is used. In this case all of the members of the final generation are at the peak and hence is a total success. Looking at the effect other variables have, it can bee seen that not all members will ie on the maximum. This could be due to mutation in the final iterations which push the member away from the maximum (particularly visible when

a mutation value of 0.5 is used) or possibly because the members are crowding round a local maximum or minimum or even the plateau. This is partially visible when 50 members are used, this is to be expected as there are more members spread across the search space and hence are more likely to find local minima and maxima. If the mutation value were set to 0 then it could be expected that more members of the population would crowd round these maxima or minima. Looking at the training graph in figure 3.2 it is clear that, for the most part, the maximum is found within a few iterations with the function peak becoming crowded very quickly.

For this task, the differences between using Roulette wheel selection and tournament selection are not noticeable due to the non-constant results which are produced. Therefore it can be concluded that each method is equally suited to the problem although with further investigation, when the two are applied to several different functions it is possible that one may prove more suited than the other.

3.2 Task 2

The testing for this task will follow the same procedure as that followed in the first task. The only major difference is that the number of members in the population is increased from ten to fifty so that a greater sample across the search space is given initially and throughout the evolution. Therefore, the normal set of variables are:

- 200 Members
- 100 Iterations
- 0.2 Elitism Value
- 0.1 Mutation Rate
- Tournament Selection with Crowding
- No Fitness Sharing

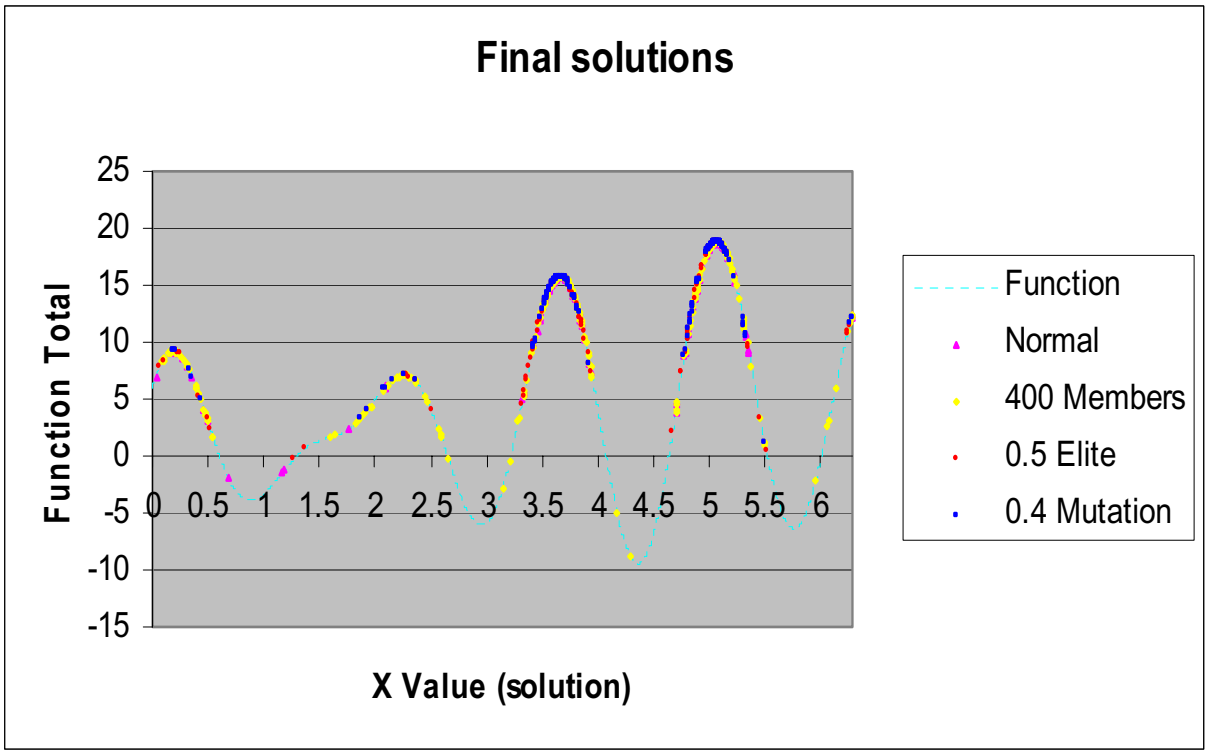

Figure 3.4 Final solutions for Task 2. The final fitness of each member of the last generation

The results show that under normal conditions the four peaks are not found in a reliable fashion, particularly noticeable on the second peak. Adding more members does provide more reliable results but it will also provide more erroneous results. It actually appears that increasing the mutation rate of the algorithm actually gives the better result. This was somewhat unexpected but can be explained as mutation will simply push a member from its current solution to a totally different solution, allowing that particular function to find its local maxima. The graph below shows what effect fitness sharing has on the function solution. Both were run with a 400 members under a mutation rate of 0.3 and an elitism value of 0.3.

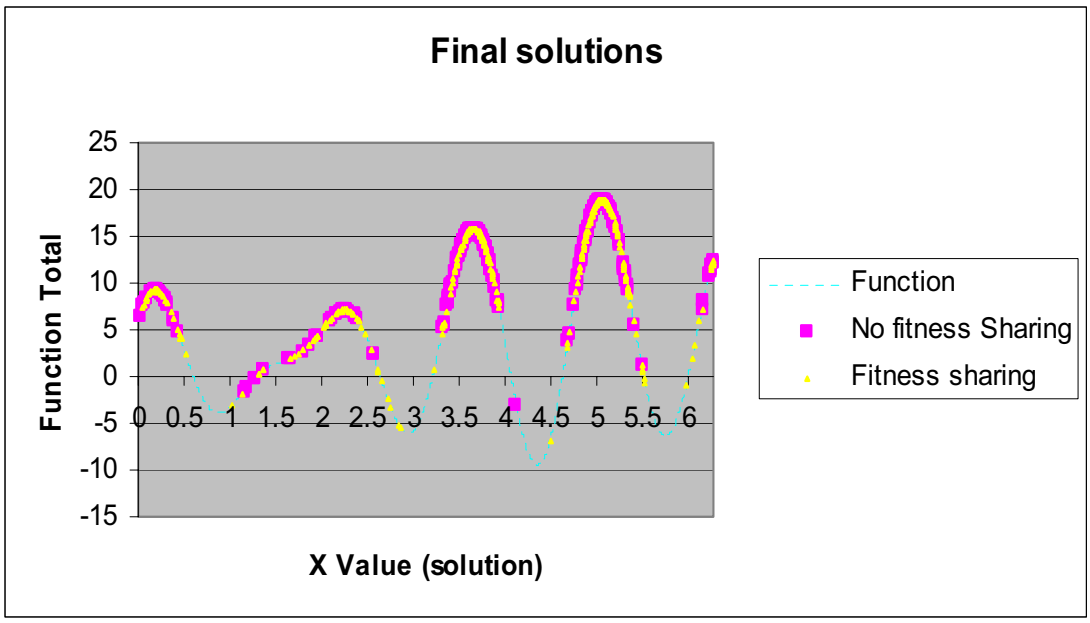

Figure 3.5 Final solutions for Task 2 comparing the crowding and fitness sharing methods. The final fitness of each member of the last generation is shown

It can be seen that the two methods produce similar results although fitness sharing will tend to give more results on the outskirts of a niche where as without fitness sharing (the crowding method) will produce more members around the maximum value. Therefore it can be concluded that the crowding method is more suited to this task.

In all the results for task 2 it can be seen that some members crowd towards the x value, 2*PI. This is because it is technically a peak of the function although its true maximum can not be seen.

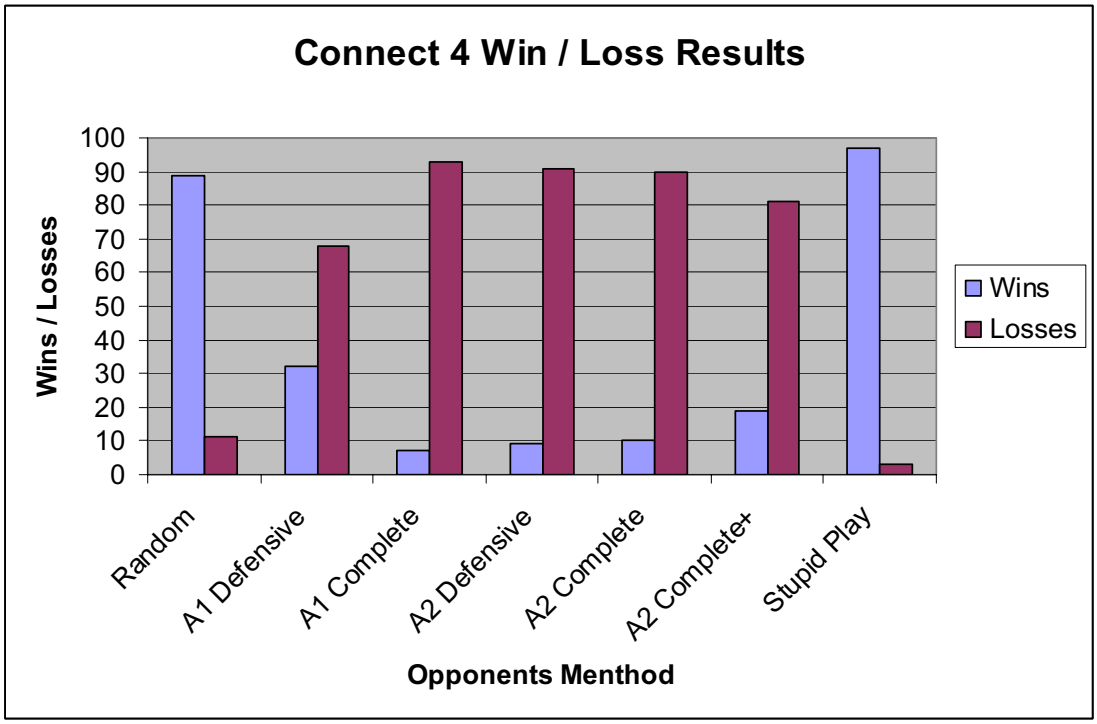

Figure 3.6 The average Win / Loss results of the final solution for Connect 4 over 100 games.

It is easily seen that the developed solution will perform well against a random or stupid opponent but far less well against an opponent with intelligence. The best results were found when the mutation level was set low (approximately 0.1 – 0.2) and the elitism value was set equally low. Actually judging against the opponent AI on any level other than the Win or Loose result is not possible as the AI methods used are not revealed. It could simply be the case that the opponent uses a game tree which can look further forward than that used however, when the maximum level which the developed algorithm is increased to a value larger than three (to 15 for example) the results do generally improve but only by a small amount (perhaps 4 more wins to losses on average per 100 games).

4. Discussion

Task one proved to be a total success. In every case it was seen that the functions maximum was reached. Obviously in the cases where there is a larger population, the more chance there is of one of the initial members to be on, or very close to, at initialisation and hence the optimum solution will be reached very quickly. As the analysis of the results show, a higher level of mutation will tend to push some members away from the optimum (as is expected as this is the behaviour required to remove members from local minima) whilst a larger elitism value (up to a certain point) will help the optimum solution to be found faster and ensure than more of the final members are positioned on the optima.

Task two also proved, on the most part, to be a success. The peaks of the function were always found however, in nearly all tested cases, the incline towards the end of the valid 'x' values was also determined as a peak. It can easily be seen that all the tested variables do produce a set of members which crowd around the peaks. Again, when a higher number of members are initially used, several of the final results are lying in random places. This could be a coincidence due to mutation. It appears that when the elitism is set to a higher value, the members become more crowded around the peaks as is expected, however the same is also true when the mutation level is increased. This is most probably because mutation ensures a good spread of the population across the search space at all stages and, as there are now more solutions than in task one, a desired solution is never to far from where a member may be mutated to. Future work in this area may include altering the algorithm such that the population is split up into niches and the member which closest match the average fitness of each niche be placed at the top of the sorted list so that it may be extracted as an elite value. Currently only the list is only sorted according to fitness and so generally only the members at the highest peak will be carried forward as an elite member.

Task two also illustrates the differences between using crowding and fitness sharing in a multi modal algorithm. The actually difference are small in this case although, as pointed out previously, the crowding methods do appear to ensure all the members are closer to the maximums than when fitness sharing is used.

Task three proved to be a partial success. When played against an opponent who places pieces randomly or the opponent was set to play in 'stupid mode', a win was almost guaranteed. However, win playing against an opponent with intelligence a win was less likely. This could simply be because the intelligent opponent looks further ahead than the three moves allowed by the current algorithm. A few basic changes could solve this but at the cost of increasing processing time. Another possibility is that fact that currently only a wining or blocking move's evaluation value can be adjusted through the genetic algorithm. A future enhancement might be to also allow the severity of a move, which is currently taken into account when assessing a moves fitness, to be evolved. Other future enhancements may include implementing a learning classifier system alongside the Genetic Algorithm in an attempt to recognise patterns in the game play to try and find the best possible next move based on previous experience.

5. References

[1] The Iterated Prisoner's Dilemma Competition, Graham Kendall, Paul Darwen and Xin Yao, http://www.prisoners-dilemma.com/

[2] Connect 4, Dan Scott, University of Reading, http://sip189a.reading.ac.uk/

[3] Imagine Cup – Visual Gaming, Microsoft Corp., http://www.microsoft.com/uk/academia/imaginecup/default.asp?s=home

[4] EvoWeb http://evonet.lri.fr/

[5] IlliGAL, Illinois Genetic Algorithms Laboratory, http://www-illigal.ge.uiuc.edu/index.php3

[6] NSGA-II, Professor Kalyanmoy Deb, http://www.iitk.ac.in/kangal/deb.htm

[7] The Learning Classifier Systems Web, http://www.cs.bath.ac.uk/~amb/LCSWEB/

[8] Learning classifier system Flash Demonstration, http://www.ai.tsi.lv/ga/lcs_maze_demo.html

[9] Genetic Algorithms : Understanding using Visual Basic., Paras Chopra , http://www.pscode.com/vb/scripts/ShowCode.asp?txtCodeId=36106&lngWId=1

[10] A Simple C# Genetic Algorithm By Barry Lapthorn, http://www.codeproject.com/csharp/btl_ga.asp

[11] Genetic Programming Inc, http://www.genetic-programming.com/

[12] Multi Niche Crowding in Genetic algorithms, http://www.personal.psu.edu/faculty/w/x/wxc28/ga/268,13,Multi-Niche%20Crowding%20(MNC)

[13] GA Methods, Tommi Rintala, http://www.uwasa.fi/cs/publications/2NWGA/node54.html

[14] A Brief Overview of Genetic Optimization, Olfa Nasraoui, Department of Computer Engineering & Computer Science, University of Louisville, http://www.louisville.edu/~o0nasr01/Websites/tutorials/GeneticAlgorithms/GeneticAlgorithms.html

[15] Evolving Strategies for the Prisoner's Dilemma, Andrew Errity, Faculty of Computing and Mathematical Sciences, Dublin City University, http://www.computing.dcu.ie/~aerrity/undergrad/prisoner/epd_technical.pdf

[16] A Generic Parallel Genetic Algorithm, Roderick Murphy, Department of Mathematics, University of Dublin, http://www.maths.tcd.ie/~rmurphy/Project/Report/report.pdf

[17] Prisoner's Dilemma Code, Idea from Scientific American, June 1995 (vol.272, no.6), Adapted by Alexander Mieczyslaw Kasprzyk, http://www.kasprzyk.demon.co.uk/www/Dilemma.html

[18] Prisoner's Dilemma, MASSACHVSETTS INSTITVTE OF TECHNOLOGY, Department of Electrical Engineering and Computer Science, http://sicp.csail.mit.edu/Spring-2005/projects/project2/project2.html

[19]JCell, Christian Spieth, University of Tübingen, Germany, http://www-ra.informatik.uni-tuebingen.de/software/JCell/tutorial/ch03s05.html

[20] On-Line Bibliography on Learning Classifier Systems and Genetic Algorithms, Andrea Murru, http://web.tiscali.it/LCS/

[21] The Prisoner's Dilemma: Technical Notes, http://www.monpetitcoin.com/technique/java/prisoner/prisonerTech.html

[22] Spatialised Prisoners Dilemma, http://www.sunysb.edu/philosophy/faculty/pgrim/applets/Prisoner.html

[23] Evolutionary Computation Research, Andrew Errity, http://www.computing.dcu.ie/~aerrity/undergrad.htm

[24] Undesirability in the spatialised Prisoners Dilemma: Some Philiosophical Issues, Patrick Grim, http://www.sunysb.edu/philosophy/faculty/pgrim/SPATIALP.HTM#S2

[25] Learning to Play Connect 4 using Genetic Algorithms, http://www.cs.rpi.edu/courses/fall02/ai/handouts/assign7.pdf

[26] Connect 4 Game Project, James Mathews, http://www.generation5.org/content/2002/connect4.asp

[27] EVOLUTIONARY PSYCHOLOGY AND ARTIFICIAL LIFE, http://www.mdx.ac.uk/www/psychology/cog/psy3260/prisoner.htm

[28] The Prisoner's Dilemma and Predator-Prey Coevolution, Joel Pomerantz, Computer Science, Yale University, http://zoo.cs.yale.edu/classes/cs490/00-01b/pomerantz.joel.jmp47/Prisoner.html

[29] Machine Learning, http://www.nada.kth.se/kurser/kth/2D5362/lecture6.pdf

[30] To Cooperate or To Defect: That’s the Prisoner’s Dilemma
Yüce Tekol, Eastern Mediterranean University, Computer Engineering Department, http://www.computer.org/students/looking/2003fall/A6.pdf

[31] Evolution of an Iterated Prisoner’s Dilemma
Strategy Using a Genetic Algorithm, Department of Scientific Computing
University of Salzburg, Richard Brunauer, Andreas Loecker, Gerhard Mitterlechner, Hannes Payer, Helmut A. Mayer http://student.cosy.sbg.ac.at/~hpayer/files/paper.pdf

[32] What is Evolutionary Programming? David Beasley, http://www.faqs.org/faqs/ai-faq/genetic/part2/section-3.html

[33] Evolution Strategies, http://lautaro.fb10.tu-berlin.de/intseit2/xs2evost.html

[34] Evolution Strategies, Eiben & Smith, http://www.cs.vu.nl/~gusz/ecbook/slides/256,1,Evolution strategies

[35] Evolutionary Alogrithms http://www.cs.nott.ac.uk/~gxk/miu/317,9,Evolutionary Algorithms - How They Work

[36] Learning Classifier Systems, http://www.answers.com/topic/learning-classifier-system

[37] Genetic Programming, http://www.answers.com/genetic%20programming

[38] The Genetic Programming Notebook, http://www.geneticprogramming.com/

Appendix I

Code Removed

The Source code has been removed to prevent plagiarism.