Evolving Ghosts in Pacman Using Evolutionary Algorithms

A 3rd year project report by
James Hume

Supervised by
Dr. Paul Goldberg

THE UNIVERSITY OF WARWICK

Department of Computer Science

2003 – 2004

Copy 1

Abstract

This project investigates the use of Evolutionary Algorithms (EA)[i] for strategy development in a pursuer-evader (PE)[ii] scenario occurring inside a maze environment, based on the arcade game Pacman.

The aim is to use evolution to develop pursuer strategies in search, pursuit, and evasion (the latter being a property peculiar to this game). Different strategy representations are used in order to compare their effectiveness.

The project also develops an evolutionary toolkit for general use with such algorithms, such that it provides a versatile "engine" for evolution. This toolkit is used to provide initial, enlightening, investigations into this species of algorithm.

Keywords

Genetic Algorithms, Evolutionary Algorithms, Genetic Programming, Pacman, Pursuit-Evasion Strategies

Document length: 14460 words.

[i] **EA stands for "Evolutionary Algorithm*(s)*". GA stands for "Genetic Algorithm*(s)*".** The reader of this report should be aware that these two terms will be used interchangeably here, although there is a subtle difference between the two. However, both share the same basic philosophy of Darwinian evolution, and since the development of GA is an extension of the EA approach to search and optimization, the distinction between the two is now almost invisible. For the interested reader, the main difference is the introduction of the "crossover" operator (introduced later in this report) in GA and some very subtle variation in selection procedures generally used between the two approaches.

[ii] "PE" will be used as an abbreviation for "pursuer-evader" throughout this report.

Contents

Table of Figures

Author’s Assessment of the Project

What is the technical contribution of this project?

A modular library for general use with EA along with a modular visual maze component capable of re-use for any maze based problem. These two components are then used in a program which combines them to produce the evolution of the pursuers in the problem. Finally, a program with the ability to load ghosts which have been evolved for play against a human opponent is also available.

Why should this contribution be considered either relevant or important to computer science?

Evolutionary Algorithms are an important tool in search and optimisation problems, key in many subject areas, especially Artificial Intelligence (AI), with wide ranging applications in the real world [2].

Investigation into their various forms and suitability in any problem domain constitutes a relevance to Computer Science. The subject is a well developed area and so this project is contributory in the sense of highlighting the use of such algorithms in an interesting problem, providing a good concrete example of implementation on a non-trivial problem. The results and conclusions of the project can be seen as a guide to the reader on the applicability of EA and the forms of solution-encoding available to the potential user, in similar problem domains.

How can others make use of the work in this project?

There will be two main uses that others can derive from this project:

1. A deeper understanding of how evolutionary algorithms can be used. Such an example of a concrete implementation should prove useful in extending ones understanding from basic introductory problems, used to demonstrate EA, to a more complex example where issues such as encoding strategy or solutions in general, become less obvious.
2. The ‘Evolution Engine” (see later) will prove a very useful toolkit for anyone wishing to investigate and learn about the functioning of EA, as well as being useful for a serious developer who wishes to harness the power of EA in their

programs without the need to develop any evolutionary framework: it is a ready to use, versatile, extensible and well documented (see appendices) modular library (implemented in Java).

Why should this project be considered an achievement?

The work involved in the research and development, planning, design and implementation constitutes a personal achievement for the author, especially as it offered an opportunity to investigate, in depth, a subject area previously unknown to the author, which also complements his course modules, such as AI.

What are the weaknesses of this project?

Initial research could have been ordered in a more optimal manner. The order of research resulted in some of the initial investigations not being used. PE papers were investigated first. However GA should have been investigated before hand, as without the understanding of this vehicle for evolution, the PE research was not applied to the full extent it could have been[iii]. A positive result for the reader can be gained here: The author would recommend investigation of GA basics and then strategy encoding (reference [1] will prove very useful for this) before addressing specific problems to anyone considering the use of GA for the first time.

[iii] A more powerful probabilistic model [11] could have been used in place of the position evaluation model. However, as it was complex, time was not available for its implementation given that the authors understanding of GA had to be developed further at the early stage in which it (the probabilistic model) was being considered. At the time of writing this report, the implementation of such a model with respect to GA optimization is understood and could be achieved in further work. For elaboration see the referenced article on a probabilistic pursuit and evasion model and suggestions for further work.

Pacman: The Game and the Problem

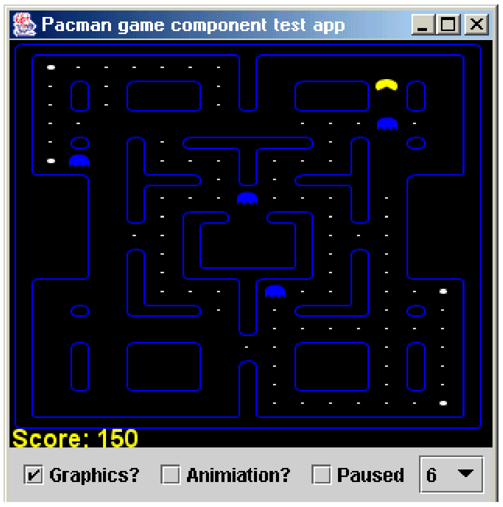
Pacman is the game used as the vehicle for the PE problem which this project addresses. The game consists of one evader, the Pacman, and four ghosts who attempt to chase and catch the Pacman through a maze.

The Pacman eats food pills located around the maze. He wins by eating all the available food pills. The ghost's aim is thus to catch the Pacman before this event occurs and also to minimise the number of pills eaten before capture.

The problem deviates slightly from a classical PE problem because in this game the hunters may become the hunted over short time durations. This occurs when the Pacman eats a "Power Pill" which will make the ghosts temporarily "edible". When this is the case the Pacman can catch the ghosts, which "die", but reappear immediately at some default, constant, location in the maze.

Through the use of Genetic Algorithms (see footnote i on pg. 2), the project aims to develop ghosts that primarily learn to play the game. Thus ghosts should learn that they need to chase the Pacman when they are invincible, but run away from him when they are edible. The ghosts also need to develop reasonable search strategies. They should not make redundant searches; for example, searching a path in the maze when another ghost can be seen to already be searching down that particular path is undesirable behaviour (unless the Pacman is in view). Ghosts also need to develop chase strategies that are not redundant. For example, all four ghosts chasing the Pacman in a line is undesirable as more effective strategies will involve flanking manoeuvres which will require some form of communicative behaviour.

The figures below provide two screen shots of the game (taken from the project's maze component developed in Java). They help to demonstrate the structure of the game and the type of maze used. Figure 1 shows the ghosts fleeing when they are edible. Figure 2 shows the ghosts chasing the Pacman.



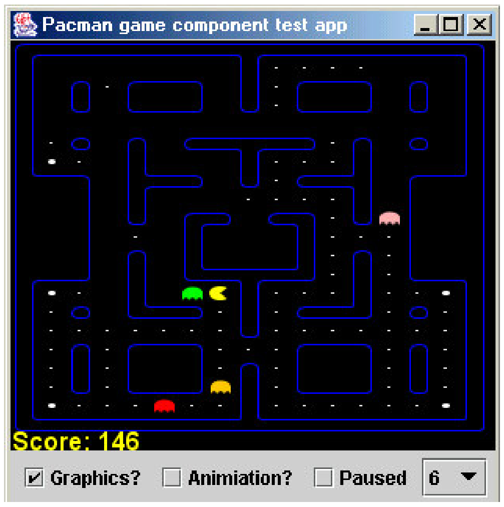


Figure 1 - Pacman is seen chasing an edible (blue) ghost.

Figure 2 - Pacman is seen fleeing invincible ghosts, whilst consuming food pills.

Having introduced the game and problem this report will now document the stages of the project in chronological order from the research into the problem, through the design and implementation phases and resulting deliverables, to the results and final conclusions.

The next section will thus document the research made, by attempting to present an introduction to GA theory in sufficient depth to arm the reader with enough knowledge of the subject area to be able to progress through the rest of the report.

Research into, and the Theory of, Genetic Algorithms

This section of the report will introduce Genetic Algorithms as a technique for search and optimisation to the reader. This embodies the initial stage of the project, bar investigations into PE problems. The reader should note that as well as theory, a lot of the terminology peculiar to GA is introduced here.

An Overview of Sources Used

Three main sources were used for learning the theory in to EA/GA:

1. Genetic Algorithms in search, optimisation and machine learning. [2]
   - This book provides in depth theory and analysis of GA, looking the mathematical theory underlying the basics, through to GA applications and the more complex representational schemes and operators.
2. How to solve it: Modern Heuristics. [1]
   - This book provides an overall, enlightening, explanation of EA. Especially useful is its discussion of how problem solutions can be represented. Useful analysis in the context of various non trivial problems is also made.
3. An introduction to genetic algorithms. [3]
   - A gentle introduction to the subject providing a good overview of GA and discussion of their applications. Also introduces GP (Genetic Programming) and makes good discussion of the various evolutionary factors which effect evolutionary algorithms in general.

Extensive use of Web resources was also made in investigating representation in GA as well as GA application to this specific type of problem. Many articles can be found using "Cite Seer" (http://citeseer.ist.psu.edu/) or the "Google" search engine (http://www.google.com).

Evolution as a Darwinian Process

The general philosophy of EA is to maintain a set of solutions to a problem, and then to compete these solutions against each other. Taking its inspiration from nature, the "best" solutions are kept and the "worst" ones are discarded. From the "best" it generates other potential solutions, such that it keeps the resulting future set of solutions (called *populations*) at least as large as the original set. By this process of "survival of the fittest", the algorithm searches through what is called a "fitness landscape", to eventually converge on a global, near-optimum solution. One can imagine the *fitness landscape* as a graph or mapping relating problem solutions to a numerical measure of performance.

The maintenance of a population of solutions is one of the GA methods of avoiding local optima. This ability has contributed greatly to the rising popularity of GA; although they might not discover the absolute optimum solution, such as a hill-climber might, they do not suffer from getting stuck at local optima or plateaus. The maintenance of several solutions also makes GA a robust search-technique in noisy environments. By avoiding reliance on a single solution from which to generate successors, which could even be an incorrect solution given noise disturbance, and being able to search "… for new solutions using the diversity of the population …" [1], GA are more likely to generate near-optimal solutions to problems where the performance measurement is subject to noise (such as this PE problem). The maintenance of a population of solutions also means that it is not just like running several searches in parallel which could otherwise have been run serially: the solutions maintained in parallel effectively share information because when one solution is significantly better than the others, it can direct the population of solutions in its direction. [4]

How GA Work

As mentioned, a GA uses a population of solutions from which to generate new, hopefully "better" solutions. However, we need to be able to represent these solutions and discover how GA can treat these representations in a manner which is completely independent of the problem domain. We also need to discover how we designate a solution as being "good" or "bad" and how this allows us to generate new solution sets (this process is referred to as creating new *generations*).

Genetic algorithms operate on a population of strings which are coded to represent some underlying parameter set [2], i.e. a set of "things" that we wish to find optimum values for. This typically means representing a solution as a binary string, but it could be as a string of integers, for example. David Fogel et al. [1] make an excellent and revealing discussion on how many archetypal problems can be represented. The representations used in this project will be discussed in greater detail at the appropriate sections.

The jargon used in GA "talk" is as follows. The string itself is called a *chromosome*. Each digit in the chromosome is called a *gene*. Each gene can assume a range of values from a domain. So, for example, in the binary domain a gene can only assume a value of zero or one. These values are called *alleles*. Each solution represented by a chromosome is called an *individual*, and each individual as well as having a *genotype*, the set of qualities resulting directly from its chromosome, will also have a *phenotype*, which is the set of qualities associated with the individual due to it's interaction with it's environment.

Once the representation has been decided on, a method of evaluating the solution is required. This involves mapping each solution to some real number which must be non-negative. This value is called the solution's *fitness*, the mapping from chromosome to fitness being achieved by an *evaluation function* which "…is your sole means of judging the quality of the evolved solutions … [barring a co-evolutionary model]" [1]. The evaluation function should give the highest fitness scores to the best solutions, showing a strong relation between what the designer views as a "good" solution and the value assigned to it via this function [1].

So, assuming that one has a set of n solutions, each having an associated fitness value, the GA can begin to form the next generation of individuals. It does this by using an uninformed search, generally taking a stochastic yet fitness-proportional sample of the population which it then uses as *candidate* solutions from which to produce the next generation. Fitness-proportional selection means that fitter individuals are more likely to make a contribution to the next generation so that the fitter a solution, the greater its contribution, in terms of number of copies, in next generation will be. The classic mechanism for the selection process is Roulette Selection. Other methods exist and are designed to improve on Roulette. They will be discussed briefly in the chapter "Investigations in to GA and Evolutionary Variables".

Once candidate solutions have been selected, they are copied into a “breeding pool” where they are randomly paired. Then two operators are applied to each pair to create the next generation of solutions:

1. Crossover:

   A method introduced to EA by the development of GA. Before its introduction EA tended to use mutation (see below) as the main form of search. Crossover is applied with a certain probability. When it is not applied, the two parents are simply copied into the next generation. When it is applied however, portions of the parent strings are interchanged.

   This interchanging of portions of the parent chromosomes provides a useful information exchange between individuals in the population, which can be seen as the exchange of “ideas” represented in various sections of the parent chromosomes [2]. This also acts as a form of global search (i.e. search in a region of the search space that is *not* “near” the two particular points defined by the solutions represented in the two chromosomes in question [1]).

   There are many types of crossover, developed to suit various problems. The simplest is single point crossover where two chromosomes are chosen as parents and a random point is selected such that they are “cut” at some point which lies within each string. The end portions of the two strings are then swapped to produce the off spring, as shown in Figure 3.

   $$\begin{array}{lcl} 100101 \mid \mathbf{00101} & \Longrightarrow & 100101\mathbf{00110} \\ 101010 \mid \mathbf{00110} & & 101010\mathbf{00101} \end{array}$$

   Figure 3 - Example of simple crossover

   Other forms of crossover exist. Most common are multiple point crossovers where more than one crossover point is defined and sections rather than halves are exchanged between strings. Depending on the problem domain even more crossover techniques exist including knowledge augmented crossovers, restricted crossover, and reordering crossovers, useful in permutation problems, such as PMX (Partially Matched Crossover). See reference [2] for further details.

2. Mutation:
   Mutation is applied to each gene in each chromosome which enters a newly created generation with a certain (normally very low) probability. When applied to a gene, it assumes a different allele, selected randomly, from the allele domain (this is just a logical bit negation in the binary domain). Mutation acts not only as tool for local search, but also as a mechanism for maintaining diversity in a population to prevent stagnation. This occurs where the majority of solutions maintained are similar, such that no more useful search using the evolutionary process can be achieved.

Niching and Diversity

This aspect of GA theory is introduced here, although it is not really material generally explained in an introduction to GA. It must however be presented as it will be referenced later on. A significant problem with some GA is that a high diversity in the population is required to enable effective search. Diversity is especially important in noisy environments. Niching and mutation are two ways of helping to maintain diverse populations. Niching is also used when the fitness landscape is multimodal where some optimal peaks may be of equal height. This process then allows solutions to be found on all or some of the optimal peaks, instead of converging to just the one.

Fitness sharing, a form of niching, simply penalises solutions which are to numerous. Thus one super solution is far less likely to get a foot hold in a generation, as its fitness would be scaled down by an amount proportional to the number of individuals in the population which embody this particular solution.

Two other popular techniques are De Jung crowding and Cavicchio’s preselection. Both these methods involve an intermediate population which thus makes the evolutions process twice as long. In both methods an original population is maintained and evaluated. The next generation is created as usual. This generation is then evaluated and the fitter individuals from it then replace their peers in the original generation. By only being allowed to replace strings which they are similar to in some way, diversity is maintained. The replacement procedures differ as follows:

- In preselection, individuals in the intermediate generation may only replace their parents in the original generation if they are fitter.
- In crowding, a group of size CF (Crowding Factor) is selected from the original population. The individual from this group which is most similar to the selected individual from the intermediate population is replaced.

This concludes the simple introduction to the way in which GA work. The report will now continue to discuss the evolutionary framework designed to support such algorithms in all their diverse forms and then to investigations made with the use of the aforementioned toolkit.

The Evolution Engine

The Evolution engine is implemented in Java, but could be easily ported to any other object orientated language. The framework is abstract yet functional enough to allow the user to simply be able to specify a decoder and evaluator (having interfaces “IEvaluator” and “IDecoder” respectively) which can then be passed to the Evolution Engine. Then the population strings can be initialised, either by the user or automatically. The user of the framework can then just request new generations without having to worry about the details of how the evolutionary process is working.

As indicated, this is meant to be a very general framework to support as wide a range as possible of genetic algorithm implementations. The "plug-in" decoders/evaluators are what give the user the ability not just to use simple GA but to incorporate ideas like fitness scaling, multi-parameter & mapped coding and more. The ability to extend from the Chromosome class allows users to create their own Chromosome structures other than the standard string or tree structures provided, but still leave all the evolution work to the evolution engine. For example, this extensibility would easily allow more advanced forms of crossover and other recombination operators to be implemented. The basic kinds of representation as strings and trees are pre-supplied for specialisation.

The results of evolution are saved in CSV-files (comma separated value files), which can be read by almost any spreadsheet package. A detailed report is produced along with a summary of the process. The engine was designed using UML; see Figure 4. For Java Documentation (JavaDoc) please refer to Appendix A: JavaDoc for Evolution Engine and Pacman.

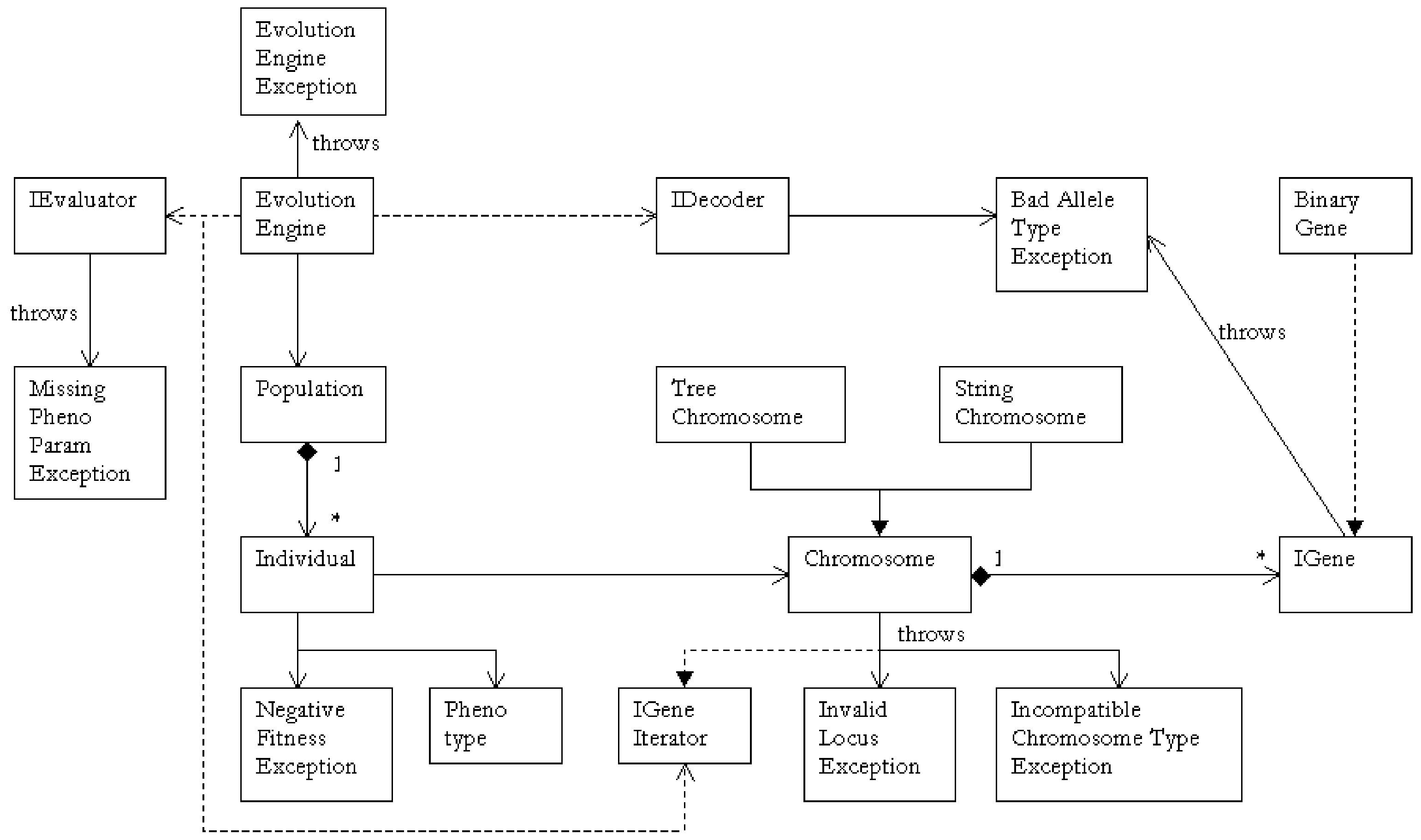


Figure 4 – The Evolution Engine design

Investigations in to GA and Evolutionary Variables

Using the Evolution Engine, investigations into two "toy" problems were made in order to gain a greater understanding of the evolutionary process. Both problems "plugged in to" the Evolution Engine with ease, requiring minimal coding to implement.

The first problem was the optimization of the function $f(x)=x^2$ as found in Goldberg [2], where only positive x values are considered. The second was a "path finding" problem where the GA had to converge on a solution which followed a specific path correctly. The alphabet was defined as A = {'N', 'E', 'S', 'W'} representing the directions north, east, south, and west respectively. A path of 30 such moves was specified as the goal and a GA was used to try and find the correct path from a randomly initialised population.

The effects of varying evolutionary parameters such as the selection method, selection pressure, and crossover and mutation probabilities will be discussed in separate sections. These problems were investigated over small population sizes which were used for the reason that if large populations were utilised, the initial random search becomes quite likely to "stumble" across pretty good solutions, rendering evolution useless. Indeed if "large" population sizes (e.g. larger than 200 – small in terms of most real evolutionary problems) were used, the effect of evolution is almost invisible, masked by the success of the initial random search. This general observation allows one to formulate an initial general rule of thumb: the larger the population size the more successful and efficient (in terms of speed of convergence on to a solution) the search may become.

Where one problem produced more noticeable results that the other, only that one will be used. All results represent the *average* of at least three evolutionary runs.

Mutation Rates

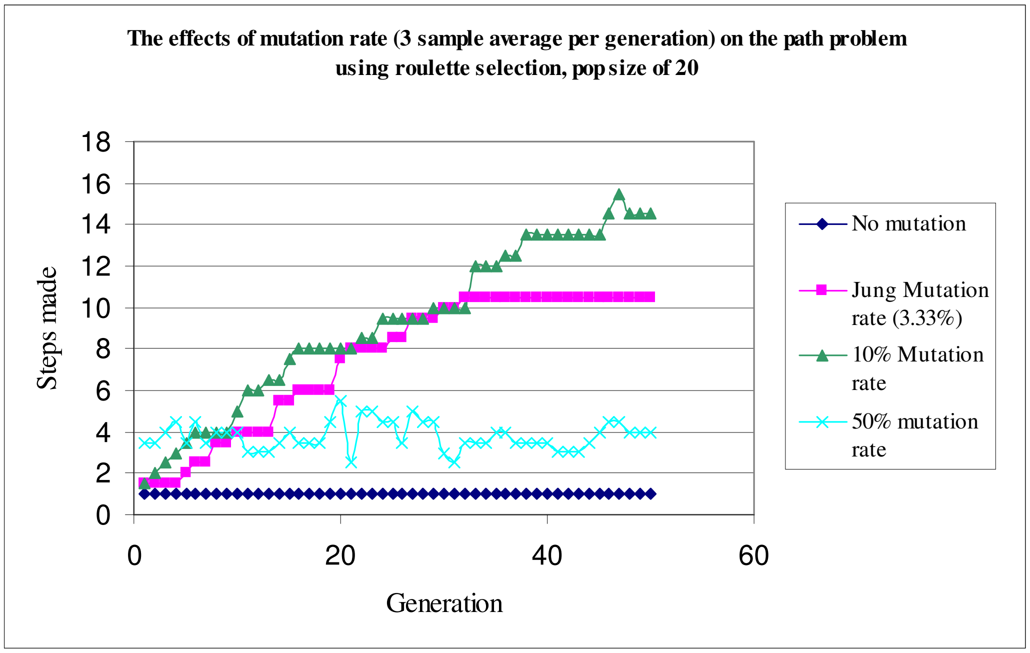


Figure 5 – The effect of mutation on the 'path problem"

The effects of mutation were far more pronounced on the path problem. As can be seen from Figure 5[iv], zero mutation and mutation rates higher than or drawing near to 50% perform very badly. However, mutation rates near Jung's suggested value [2] perform well on this problem. A higher rate of 10% increases performance greatly; however on the first optimisation problem it degrades performance slightly. This suggests that mutation is a more effective tool for search in high cardinality alphabets (an alphabet's cardinality being the size of the set of symbols over which it is defined. E.g. the binary alphabet has a cardinality of two). This intuition can be backed up by the observation that crossover becomes a less effective tool for search as it cannot search 'between" alphabet symbols i .e. if two parent strings are NS and SE, crossover can only generate NE and SS. If, for example, we represent north as 00, south as 01, east as 10 and west as 11 we now have:

NS = 0001, SE = 0110

[iv] The graph shows the number of steps made by the fittest solution along the goal path in each generation averaged over 3 trials. For example if the goal path is "<u>NESW</u>SEW" and the fittest solution in a generation is "<u>NESW</u>WES" then it is said to have made 4 steps along the goal path.

Crossover is then able to produce the following solutions:

Crossover point = 1, crossover point = 2, crossover point = 3

0001 = NS, 0110 = SE, 0010 = NE, 0101 = SS, 0000 = NN, 0111 = SW

Thus it can be seen that by using a lower cardinality alphabet, crossover as a search technique would have been more productive. The fact that it can produce more solutions given a smaller alphabet also has links to schema theory as explained in Goldberg 1989 [2].

A lower cardinality alphabet will require longer strings meaning that there will be more possible similarities between strings in the population for the GA to exploit. If an extra, "don't care" symbol, '*' is introduced to the alphabets, a pattern match can be done against each string. One can create a set of patterns, like simple regular expressions, which match one or more "real" strings. Goldberg shows that each pattern (called a *similarity template* or *schema*) which has an above average fitness will receive exponentially more contributions in coming generations. Thus the more of such similarity templates available, the more "ideas" of what makes a fit string are available for exploitation. For example, if the goal path is NESW, it can be represented by the string "N E S W" or "00 01 10 11". We could dictate that any path matching "NE**" is quite fit. However in the lower cardinality alphabet any string matching "000*****", "00*1****" or "0 *01****" would have an approximately equal fitness as they are "getting close" to what the first half of the solution must be . Thus because more fit schemas can exist, more strings can match them and be selected as candidates for reproduction and crossover (generally referred to as recombination), then being used as a basis for further "promising" searches for future generations.

Crossover Rates

This section demonstrates the effect of varying the crossover rate (i.e. the probability with which crossover is applied), but also acts as an argument for mutation as a diversity maintainer.

The results obtained using crossover with no mutation on the binary and path optimisation problems were in some ways inconclusive as all of the searches at all rates of crossover performed very poorly on average. This is, however, a persuasive argument for mutation as a search and diversity maintenance technique. The reason that crossover did not work well was that the populations stagnated very quickly. The tests also served to demonstrate the obvious: that with no mutation and no crossover evolution could not occur (as the GA effectively has a null successor function in this case).

Using the Jungian mutation rate improves the search (as shown in the above section on mutation). The effect of crossover on both problems was still debatable; varying the probability from zero to one produced no conclusive benefits. The effect of crossover rates on the project's performance is thus investigated further in light of these initial investigations.

Selection Methods

The basic selection method, already introduced by name, is Roulette Wheel selection. When used, each individual in a population is assigned a portion of the wheel in proportion to their fitness, relative to all other individuals. Thus a biased wheel is produced, N spins of which, generate N parents as candidates for the next generation. The main criticism of such a method is that the N spins could all be unlucky spins, destroying most, if not all of the current fittest solutions by not selecting them as candidates for the next generation. Two main techniques can be used to combat such "*stochastic errors*": either preserve a percentage of the better solutions across the generations, or use a sampling method less prone to making such mistakes.

There are many selection methods, of which only a few are touched upon here. Others not included here include deterministic sampling [1,2] truncation selection [1] (more akin to EA than GA), linear, power-law, and sigma scaling (a.k.a. sigma truncating in Goldberg) [1,3] and many, many more, each with certain advantages and disadvantages.

After looking at a sample of selection methods, this chapter will go on to compare a handful of those examined.

Selection Methods to Reduce the Potential Loss of Good Solutions

The first point, made in the introductory paragraph of this section, is hinting at the use of what is called "elitism", a method introduced by De Jung in 1975 in "… his important and pivotal dissertation, 'An Analysis of the Behaviours of a Class of Genetic Adaptive Systems'…" [2]. Elitism as commonly used, and as used in the Evolution Engine, is a slight variation on what Jung originally specified but maintains the "essence" of the principle. When a new generation is created a percentage of the current fittest solutions are preserved and copied with out modification into the next generation (skipping the candidate pool where solutions are mated and mutated). Thus from a population of N solutions, where n are preserved using this method, only N-n solutions are generated using evolution for the next generation, the remaining n fittest being copied directly through from the previous generation.

This is also related to the issue of "stochastic errors" which is discussed below. In the example shown in Figure 6, had we used 10% elitism, at least one copy of individual #9 would have survived, at least guaranteeing that the best solution is never forgotten.

Selection Methods to Reduce Stochastic Errors in Sampling

The latter point, made in the introductory paragraph of this section, is hinting at discovering ways to reduce stochastic errors. Figure 6 shows a sample from a run of the x^2 optimisation problem. It clearly demonstrates how stochastic errors have meant that the best solution from generation two has been "forg otten" by the GA in generation three. Individual #9 from generation two should contribute seven times to the next generation, but due to the unlucky spins of the roulette wheel, has in fact not been able to make any contribution.

Many researchers have attempted to reduce these stochastic errors using modified sampling techniques whist still maintaining some of the randomized element involved. "Stochastic Univ ersal Sampling" (SUS), first introduced by James Baker in

1987 [3] and "Stochastic Selection without Replacement" (SSWR) [2] are two such methods. Both methods attempt to make individuals contribute a number of samples closer to their expected contribution to the next generation. As investigations made have shown, these techniques often produce faster convergence. However the reader must note that the "toy" problems used as a test bed are not noisy, meaning that the evaluation function is always correct. Thus individuals will make contributions to future generation in proportion to their real worth. When the environment is noisy, the evaluation function may not judge a solution correctly. In other words contributions made become less closely correlated with a solutions real worth. The most extreme form of such stochastic error reducing methods is deterministic sampling.

Gen 2

#	Parent_#1	Parent_#2	Chromosome contents	Fit	Expected contribution to next gen	Actual contribution
0	2	2	FFFFFFFFFFFFFFFFFFTFFFFFFFTFFF	1.0000044	0	0
1	5	5	FFFFFFFFFTFFFFFFFFTFFFFFFFTFFF	1.0022556	0	0
2	8	6	FFFFFFFFFFFFFFFFFFTTFFFFFFTFFF	1.0000066	0	2
3	6	8	FFFFFFFFFFFFFFFFFFFFFFFFFFTTFF	1	0	0
4	7	7	FFFFTFFFFFFFFFFFFFTFFFFFFFTFFF	1.0746126	1	2
5	8	8	FFFFFFFFFFFFFFFFFFTFFFFFFFTFFF	1.0000044	0	2
6	8	9	FFFFFFFFFFFFFFFFFFTFFFFFFFTFFF	1.0000044	0	1
7	9	8	FFFTFFTFFFFFFFFFFFTFFFFFFFTFFF	1.1757485	2	2
8	7	6	FFFFFFFFFFFFFFFFFFTFFFFFFFTTFF	1.0000044	0	1
9	6	7	FTFFFFFFFFFTFFFFFFTFFFFFFFTFTF	1.7792872	7	0

Pop max fit	Pop min fit	Pop ave fit	Pop sum fit	Num muts	Num cross
1.7792872	1	1.1031928	11.031928	11	3

Gen 3

#	Parent_#1	Parent_#2	Chromosome contents	Fit
0	4	2	FFFFTFFFFFFFFFFFFFTTFFFFFFTFFF	1.0746149
1	2	4	FFFFFFFFFFTFFFFFFFTFFFTFFFTFFF	1.0011296
2	5	5	FFFFFFFFFFFFFFFFFFTTFFFFFFTFTF	1.0000066
3	7	7	FFFTFFTFFFFFFFFFFFTFFFFFFFTFFF	1.1757485
4	2	2	FTTFFFFFFFFFFFFFFFTTTFFFFFTFFF	2.3714101
5	5	5	FFFFFFFFFFFFFFFFFFTFFFFFFFTFFF	1.0000044
6	4	7	TFFFTFFFFFFFFFFFFFTFFFFFFFTFFF	3.3982234
7	7	4	FFFTFFTFFFFFFFFFFFTFFTFFFFTFFF	1.1757491
8	6	8	FFFFFFFFFFFFFFFFFFTTFFFFTFTTFF	1.0000067
9	8	6	FFFFFTFFFFFFFFFFFFTFFFTFFFTFFF	1.0366378

Figure 6 – Example of stochastic errors using the Roulette Selection method

Selection Methods to Reduce "Premature Convergence"

Premature convergence is concerned with population stagnation. This has already been described briefly. It occurs when all the chromosomes in the population are similar to the extent that no more useful solutions can be produced as all variety is lost. For example, a population containing n sets of solutions all equal to $b_1b_2..b_m$ cannot produce any more solutions via crossover. Mutation, as has been demonstrated (see Figure 5 – The effect of mutation on the "path problem"), should be at a very low rate and will thus have a very minimal effect. The GA can not make useful progress after this point. Many of the selection techniques discussed so far do not address this issue; if the initial population has a few super-strings, they will dominate the second generation.

One would, however, like this kind of convergence at a late point in the evolutionary process, as eventually a solution must be found. However, doing this too quickly can mean that the algorithm converges to a potentially poor local optimum, or even worse, just some arbitrary point in the search space. At the start of the evolution we want to *explore* the search space, making best use of diversity to generate solutions over as much of it as possible. Then, near the end of an evolutionary run, one best solution should be *exploited*. This means that the GA should investigate only solutions within its neighbourhood, effectively forgetting other less fit solutions. The general point is that one must not sacrifice exploration for exploitation too early [2].

So how does the GA decide when to explore and when to exploit, and at what rate? The decision is made by the *selection pressure*, which is the factor that determines how significant the "gap" between good and bad solutions is. The effect of selection pressure applied in a constant fashion over all generations in the path problem, as seen in Figure 7, shows how it can affect an entire run of a GA and its success. This demonstrates that selection pressure can be more influential than selection technique. Such pressure differences are applied in the evaluation function and never change. However, various selection methods seek to vary selection pressure during an evolutionary run.

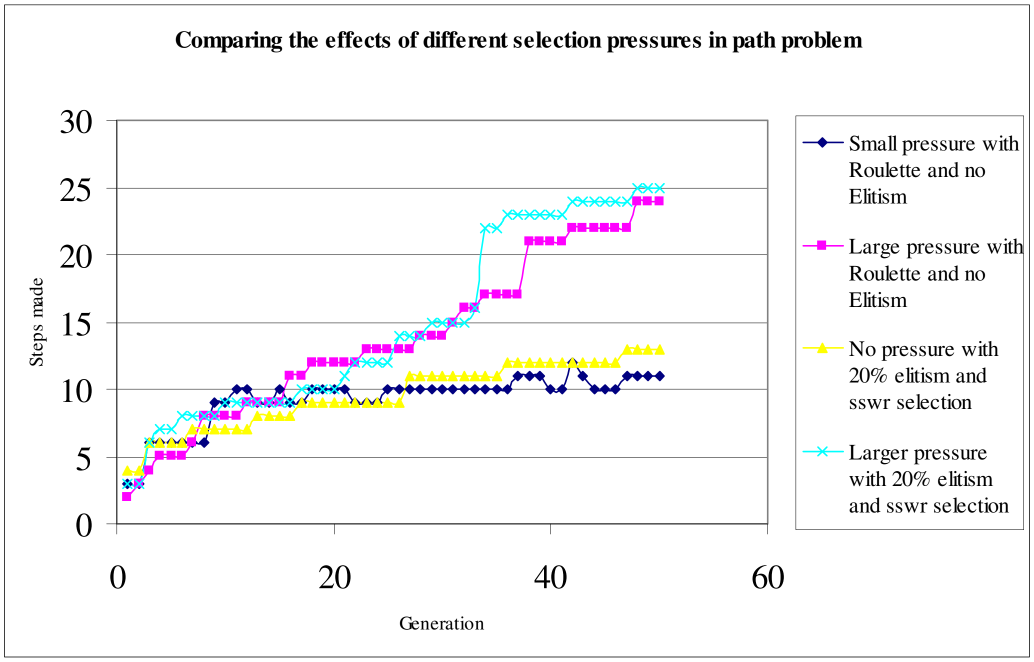


Figure 7 – Selection pressure determining the GA success.

To explore, the selection pressure needs to be low, so the gap between any two solutions must appear to have little significance. When this is the case, solutions are not differentiated as being good and bad to a large extent, so all solutions have a reasonable chance of contributing to the next generations. Then later on, to begin to exploit better solutions, the gap between good and bad solutions should appear much more significant. When this is the case, the best solutions will get a foothold in the next generations and this sole set of solutions can be explored on a local, exploitative, basis. Methods using this technique are generally referred to as scaling mechanisms and adjust the fitness values for each generation based on the generation number. The general idea is to "scale back" the fitnes s function earlier on so that super-strings do not appear so fit relative to the population average, and thus do not get a dominating foothold in the next generation. Scaling mechanisms are not investigated here.

One popular method used to avoid premature convergence, developed by James Baker in his 1985 paper "Adaptive selection methods for genetic algorithms", was rank selection. Individuals are sorted in order of fitness and are assigned a rank. They receive a number of copies in the next generation in proportion to their rank. Another selection method is tournament selection [3]. This method selects two (or more) individuals from the population at random, and selects the fittest based on a biased coin toss.

The two methods outlined above have the problem that selection is no longer fitness proportional. This is the main criticism levied at them. Such criticism can not be levied at scaling mechanisms (e.g. Sigma Scaling).

A Comparison of Selection Methods

A handful of the selection methods described in the previous sections are now compared. Figure 8 and Figure 9 make a comparison between the standard Roulette Selection, stochastic error reduction and premature convergence avoidance techniques. Both figures show very clear trends.

Tournament selection is quite ineffective, most evidently on the path problem. This is to be expected as the evolution is no longer survival of the fittest but survival of the luckiest since selection is no longer fitness proportional to any significant degree. Rank selection on the other hand performs noticeably better on the binary string optimisation problem but takes longer to converge. Again this can be explained by the loosening of the fitness to selection-probability coupling that rank selection introduces. In the path problem Rank selection fails to perform well, although it still does slightly better than Tournament selection. Due to the looser coupling of fitness to selection-probability Rank selection more noticeably is not able to maintain and steadily build on better solutions when it finds them. This is most evident in Figure 9 where a plateau of eight steps is maintained but, not only is it not built upon, it is then lost. It appears that for higher cardinality alphabets fitness proportional selection becomes a much more significant requirement.

SUS and SSWR both out perform Roulette selection, by stimulating faster convergence through stochastic error reduction, in this noiseless environment. Given this environmental property one would not expect the results comparing error-reducing techniques to apply exactly to Pacman as noise obfuscates the value of solutions given by the evaluation function; thus maintaining the best might sometimes be maintaining the mediocre or, even more unpleasantly on occasion, the worst.

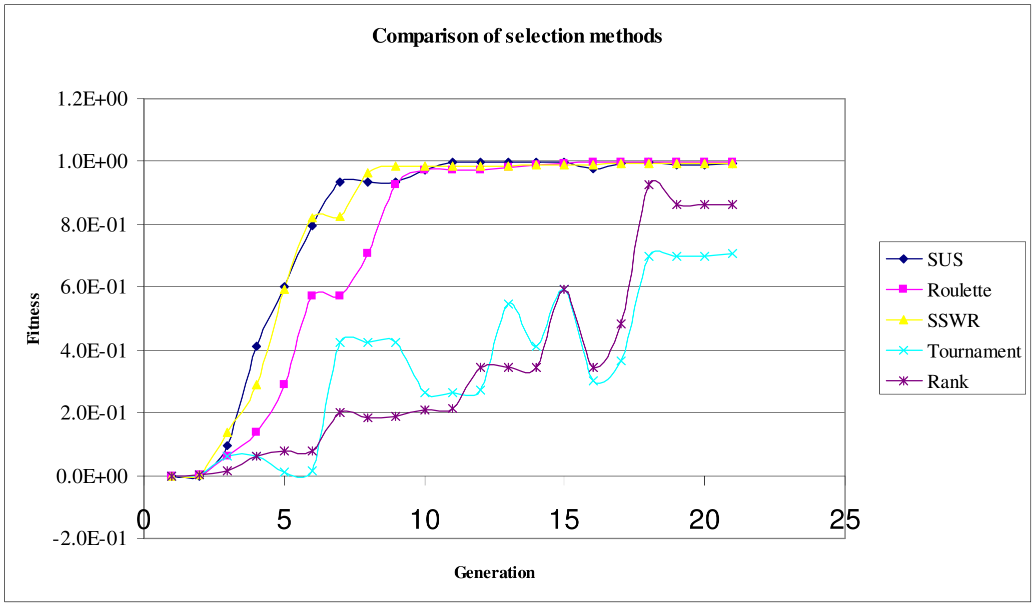


Figure 8 - A comparison of some selection methods with the x^2 optimisation problem

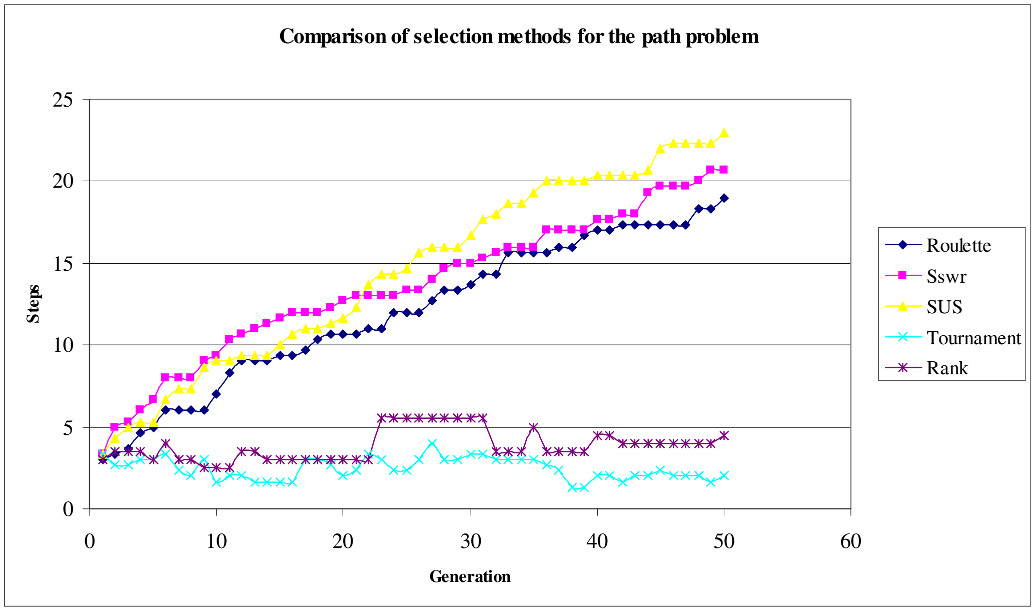


Figure 9 - A comparison of some selection methods with the path following problem

Having seen some comparison of evolutionary factors this report will now set out to describe how the application of GA to the PE game Pacman was made.

Applying GA to Pacman

The leap from "toy" problem representation, to the representation of abstract strategic ideas is not necessarily an easy one to make. It is difficult to see how the yes-no decision making one might express when discussing strategy can be encoded in a (binary) string. This was probably the steepest section of the learning curve in this project.

There are many ways to encode strategies and programs. The project author recommends "How to solve it: Modern Heuristics" [2], for a comprehensive discussion of this issue. The two methods that were decided upon for this project were the use of a binary string in a position evaluation model and a decision tree representation, an extension of GA called GP (Genetic Programming), as developed by John Koza. The latter method is a much more intuitive way to represent strategy, as it allows structured program parse-trees to be used, which are closer to the way in which one might envision a search and chase strategy. This type of encoding is attractive because, as Koza argues, "It is unnatural and difficult to represent computer programs of dynamically varying size and shape with fixed length character strings" [5].

Having decided on two representational strategies, a general model for the game is needed. The maze is modelled as a grid, where both pursuers and evaders can only see over a set distance. Apart from a central "cabin" the maze has no open areas making the only possible moves at each path square north, east, south, or west. During the game each player makes a move in turn. A pursuer catches an evader when both players occupy the same square. The Pacman plays as nature[v], and always knows where the ghosts are located. Both the Pacman and ghosts know the maze structure and state. The ghosts however do not know where the Pacman is unless he is in their sights. How ghost groups communicate this knowledge is discussed separately in the model descriptions in the sections below, suffice to say that the position evaluation

[v] A term used in many PE papers meaning, in this case, that Pacman plays in a fully observable environment and thus knows its entire state and the state of the pursuers.

model uses an idea of "implicit" communication, and the GP model uses a more "explicit" form of communication.

The binary string representation will be discussed first, followed by an explanation of the GP method used. Then having dealt with the coding used in each model, the method used for evaluating the ghosts' performances will be discussed.

The Position Evaluation Model

This model uses a form of "implicit communication" which means that when one ghost can see the Pacman, all the other ghosts are immediately aware of the location of Pacman. Thus the communication between the ghosts is implicit in the model; it is not up to each ghost to alert its colleagues to sightings as this is done automatically without the ghosts having any explicit "knowledge" of this process.

The ghost decision making is made by an evaluation of each square based on the contents of that square and its surroundings. The ghost simply moves in the most desirable direction by taking the average value of the squares to the north, south, east, and west as far as it can see (a constant number of squares). When two directions are equally desirable, one is chosen at random. The reason that an average must be taken is that there may be a different number of squares in each direction which would otherwise give an unfair bias (in general) to the directions with the most path squares along it. Wall squares are always viewed as massively undesirable (negative infinity value) and do not have their desirability affected by surrounding squares. Neither do they affect the desirability of adjacent path squares.

Food pills, power pills, other ghosts (the distinction between invincible, edible, and flashing ghosts is important), the Pacman, and prizes affect a square's value. Each object also radiates desirability, creating an area of influence. The area is defined by a range which simply specifies how many squares in each direction should have the object's desirability (decayed linearly with distance) added to their desirability value.

The problem thus becomes one of finding the correct weighting for each object in the grid, such that appropriate search and chase strategies result. Figure 11 and Figure 10 show a section of an imaginary maze. They help to explain how the idea of desirability allows to the ghosts to make correct or indeed incorrect moves, and also serve to demonstrate how implicit communication is used.

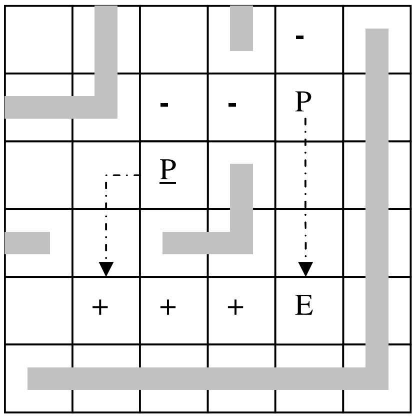


Figure 11 – Position evaluation when invincible

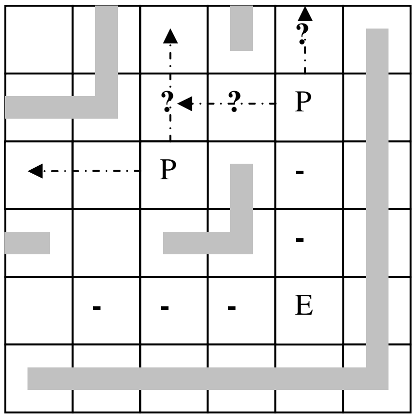


Figure 10 – Position evaluation when edible

In Figure 11 one can see how the underlined ghost (**P**ursuer) might evaluate the maze. In this scenario both ghosts are invincible and using this generalised yet idealised strategy the underlined ghost evaluates the squares near his colleague negatively. Thus, the north direction becomes undesirable when compared to the west and the ghost moves west. In his next turn, because the ghost has had the location of the Pacman implicitly communicated to him (his peer can see the Pacman), and because the squares around the Pacman are valued as being very desirable, he will move south, completing a flanking manoeuvre, capturing Pacman. This demonstrates how a correct evaluation of the objects in the maze and the form of communication used results in a rational strategy. However, this evaluation first has to be learnt by the ghosts and then refined. For example, initial ghosts might evaluate the squares containing and around the Pacman negatively.

Figure 10 shows that the grid evaluation must change when a ghost is either edible or flashing. When a ghost is flashing he is still edible; this state serves to indicate to the Pacman, who is now the pursuer, that the ghost in question will soon become invincible again. In the figure, the Pacman is evaluated as being undesirable and so the ghosts move away from him. However, initially the ghost may not know to invert his evaluation as shown, and so this will also have to be learnt.

In the preliminary investigations into this model, it was found that ghosts were not able to develop appropriate strategies when no evader was visible, as inevitably, they would move into a situation where all directions were equally desirable and just oscillate between two squares. To remedy this problem, two mechanisms were added. Firstly a new weighting was introduced which was applied to the most recently visited square to make it appear undesirable and thus bias the ghosts into moving in one direction. This weight becomes another parameter to be optimised by the GA. The second method to help encourage search was a 'history grid", which records the number of steps since the ghost last visited each square. This is then added to the position evaluation grid with the result that squares visited least recently appear more desirable. The value for any square in the history grid is capped at a certain level. This ceiling value does not appear as a parameter to be optimised.

It was also originally thought that parameters (desirability values and desirability ranges) should be allowed to vary from -16 to +16. These ranges dictate the area of influence of an object i.e. how many squares in each direction are affected. However, testing this on ghosts with hand coded weight sets (sets of desirability values to be used in position evaluation) revealed that the variation of -16 to +16 did not provide enough ability to create noticeable differences between squares of varying contents. It was decided that a variation of -255 to 255 would provide enough ability to produce effective, noticeable difference between different combinations of objects.

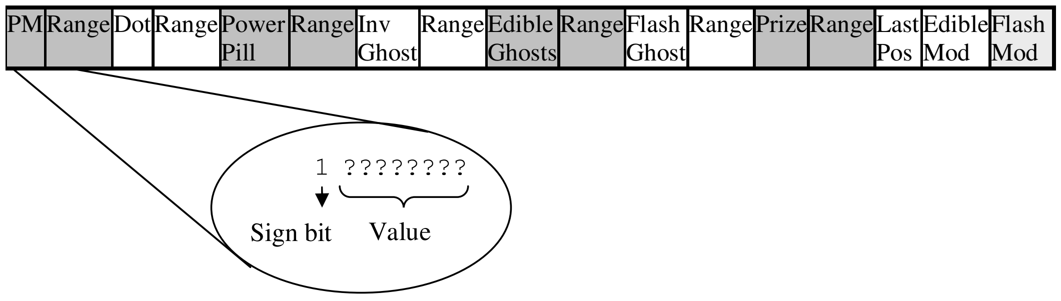


Figure 12 – Multi parameter fixed point coding for position evaluation model

Having made these adjustments to the original representation, the result was a multi-parameter, fixed point coding consisting of seventeen parameters, each 9 bits long as seen in Figure 12. The parameters 'Edible Mod" and 'Flash Mod" are used to determine how the ghosts should change their evaluation of the Pacman when they are either edible or flashing.

Next to each object's desirability value is its corresponding range parameter. The value can be negative in this scheme which might strike the reader as odd. However, a negative range is meant to have a very specific meaning. When an objects range is negative the square which it occupies has its desirability incremented by that objects weight. However the surrounding squares constituting the objects area of influence will receive a negative (decaying) desirability value. This allows the model to express the idea that one object might be more desirable than a group of like-objects. For instance a single food pill might be more desirable than a group of pills because the Pacman is likely to have to backtrack to get this sole pill.

The GP Model

This section will begin by describing the model used and the form of communication that the ghosts use. The ideas behind and reasons for considering GP will then be discussed, followed by a description of how it has been implemented.

An Introduction to the Model

This model uses a form of "explicit" communication. Ghosts only know w here the Pacman is when they can see him for themselves. Communication now takes on a more "explicit" form ; to alert his colleagues a ghost can only drop a marker serving as an indicator to the other ghosts that the Pacman was seen near the said marker recently. The sighting is known to be recent because the marker decays with time until it no longer exists in the maze. The idea of such markers, usually referred to as "pheromones" has been widely used in similar problems where some form of communication is required [6,7]. In this model the life-span of markers is set to a constant period. This form of communication was used in this model because it has a greater expressive capability; ghosts can reason as to when they should drop markers which becomes a trait that they should learn. Had this been implemented in the previous model, the knowledge of when to drop markers would have had to be implicit and so communicative behaviour using markers could not have been evolved.

A Description of Genetic Programming

GP works by using a tree of functions and terminals (the leaves of the tree). Trees can be of arbitrary size, and each tree represents a program. This being the case, a sub tree can be viewed as a function or module within the program. For sources on GP see references [1, 3, 5, 6, 8] and John Koza's website (www.geneticprogramming.org) which also provides much material on the subject. A short introduction will, however, be presented below.

A parse tree consists of function nodes and leaf nodes called terminals. Although as mentioned, the tree can be viewed as a program proper, it can also be viewed as an, "if-then" decision tree, as applied to the Snake game by Tobin Ehils [8]. This is especially appropriate for the type of strategy-evolution problem being considered. The only other syntactic requirement is, as Koza states, that "each function in the function set should be well defined for any combination of elements from the range of every function [and terminal] that it may encounter…" [5]. This requirement will be "stretched" to fit more appropriately with the semantics of decision trees in the next section.

Even with the different representation and constraints on function sets and terminals, GP works just like a GA. Crossover exchanges randomly selected sub trees between trees, thus allowing solution sizes to vary (which is not the case for GA), and mutation changes the value (constrained by domain) of random nodes. Mutation now has the option of having different effects on node types. For example it might use a Gaussian mutation for numeric terminals and random mutation for function nodes, or replace certain sub trees with randomly generated ones [1].

GP Applied to Pacman

The function and terminal sets chosen for this representation are as shown below:

Function set

- IF_PACMAN_{DIR}
- IF_GHOST{DIR}
- IF_MARKER_{DIR}
- IF_WALL_{DIR}
- IF_DOTS_{DIR}
- IF_PPIL_{DIR}
- IF_INVINSIBLE
- IF_FLASHING
- IF_EDIBLE
- IF_NUM_DOTS_{DIR}_LT
- IF_DISTANCE_TO_PM_LT

Terminal set

- MOVE_{DIR}
- MOVE_LEAST_RECENT
- MOVE_RANDOM
- DROP_MARKER
- NUM_DOTS_{DIR}
- DISTANCE_TO_NEAREST_[DOT|GHOST|MARKER|POWERPILL]_{DIR}
- DISTANCE_TO_PACMAN
- DISTANCE_TO_NEAREST _MAKER
- [0, 16)

Where ‘NAME_{DIR}” appears in a function or terminal it implie s that the name expands into four versions, one for each direction, yielding “NAME_NORTH”, “NAME_SOUTH”, and so on. There are thus thirty two functions and thirty terminals where ‘[0 , 16)” is considered as one type of terminal which can have a value that ranges from 0 to 15. There are thus a total of sixty two possible node types in a tree.

The function set is very large, and this fact raised worries before the model was even tested. In her book [3], Melanie Mitchell makes the observation that many examples of the use of GP contain fairly high level instruction sets. The relevance here is that this observation indicates that a GP model using fine-grained commands such as “*if Pacman in direction then move in some direction*”, as opposed to courser-grained commands such as “*if Pacman can be seen then follow*” might not perform well. However, it was felt that raising the function and terminal sets to a more abstract level would trivialise the problem and make it far less interesting, if indeed meaningful. It would also mean that development of certain chase heuristics would become impossible. As it stands the terminal set includes a slight compromise by including commands such as “*move (uniformly) at random*” and “*move to least recently visited square*” because these are slightly more course-grained than their peer terminals. The latter terminal also solves a similar problem to that encountered in the position evaluation model. When the Pacman is not in sight the ghosts can reason about which direction to take, but when all directions have similar characteristics an ability to search is needed. This is embodied in this one terminal.

Another point about the function set specifically that the user might have noticed is that Koza's requirement for closed function and terminal sets has been broken: the "less than" functions do not have the same domain as their peers. This makes crossover a little more complicated as it is possible for the operation to create invalid decision trees. For example: ifDistanceToPMLessThan(drop_marker,...,**move_back**) is invalid as the last operand must be a number. However, this problem can be overcome by using a knowledge augmented, greedy crossover as will be explained shortly.

Firstly a justification for this method must be made. Using GP the "less than" functions are usually used with four operands, where if "...the value of the first argument is less than or equal to the value of the second argument, the third argument is evaluated and returned. Otherwise, the fourth argument is ..." [5]. The reason for deviating from this method is that the GP tree is being treated as a decision tree and functions therefore don't really return anything, they just determine which choice is eventually made. Thus if the proposed scheme combined with normal crossover was allowed, the example given above could easily occur, and what's worse is that "move_back", which should have evaluated to a numeral value, could be an entire sub tree of commands, which could end up with non-numeric terminals. It thus becomes simpler to use a not-so-blind crossover to help maintain a nicer semantic structure to the trees and reduce complexity. The crossover used will now be described.

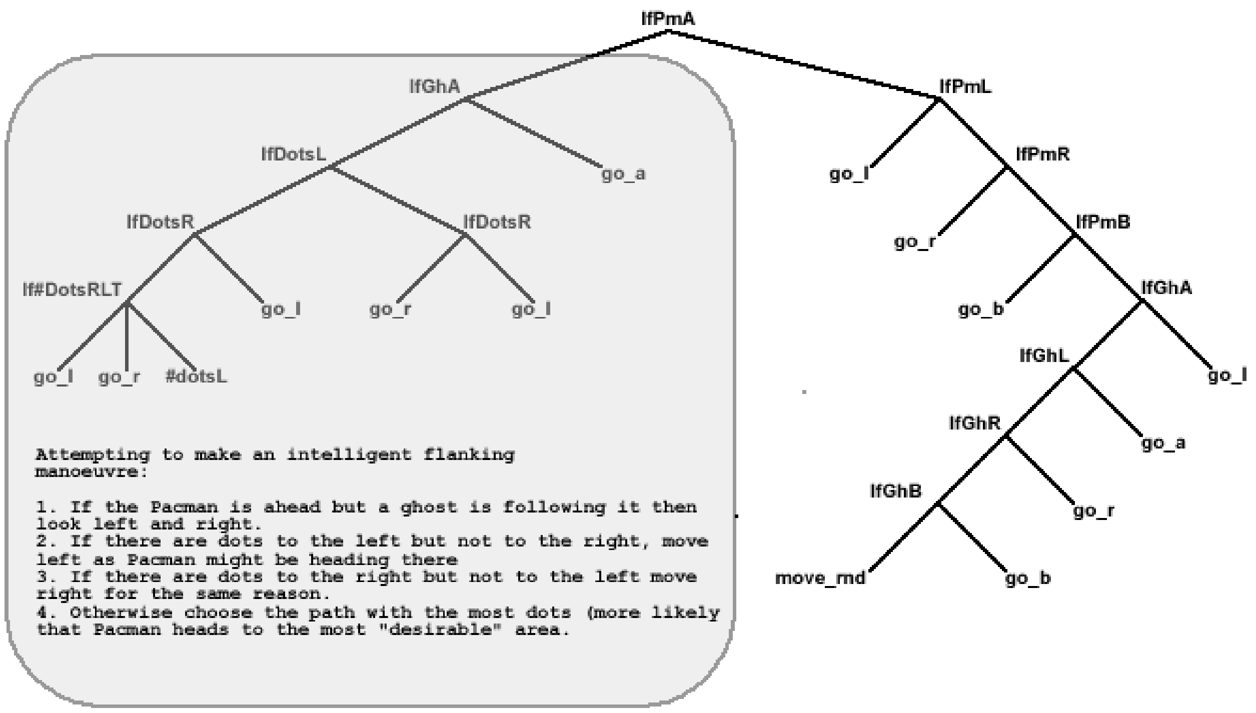
A knowledge augmented crossover is just one which has some knowledge of the problem domain, such that crossover is unrestricted yet always produces valid child chromosomes and also tries to maximise the productivity or effectiveness of the crossover. This is described by Goldberg [2] when he discusses work by Grefenstette et al on crossover and the Travelling Salesman Problem (TSP) problem.

The crossover used in this problem always produces valid offspring and greedily tries to maximise the chance of swapping numerical terminal nodes if one is selected for crossover (as these nodes should be in a small minority). The crossover is described in Figure 13.

- If the cross over point on the first parent cuts a non-numerical child off the tree
    - If the same point on the other tree also has a non-numerical child then cross over as normal.
    - Otherwise start at the top of the first tree and search for a numerical node
        - If one is found crossover
        - Otherwise go up one level in the second tree and crossover as normal
- If the cross over point on the first parent cuts a numerical child off the tree
    - If the same point on the other tree has a numerical child crossover
    - Otherwise start at the top of the second tree and search for a numerical node
        - If one is found crossover
        - Otherwise go up one level in the first tree and crossover as normal

Figure 13 – Knowledge augmented crossover used with GP model

As mentioned, this method of representation offers more expression than the position evaluated model, and as such it was hoped that the ghosts might be able to develop more complex searching and chasing heuristics. For example, now it is possible for the ghosts to ‘think” about when and how to execute a flanking manoeuvre; a ghost might decided that if there are ghosts ahead of him, but the Pacman is also ahead, he should then, at the next junction, move in the direction in which the most food pills exist and no other ghosts can be seen, under the assumption that the Pacman will move towards areas of the maze with the highest concentration of pills. This is shown in Figure 14. Using this expressive capability even more advanced strategies, it was hoped, would evolve to forms such as the one shown in Figure 15. Here we see a combination of the previous flanking example (demonstrating how crossover might serve evolution by combining useful functions or notions about strategy); it produces a ghost which will chase effectively but also, when edible, flee intelligently.



```
Attempting to make an intelligent flanking
manoeuvre:

1. If the Pacman is ahead but a ghost is following it then
look left and right.
2. If there are dots to the left but not to the right, move
left as Pacman might be heading there
3. If there are dots to the right but not to the left move
right for the same reason.
4. Otherwise choose the path with the most dots (more likely
that Pacman heads to the most "desirable" area.
```

Figure 14 – An example flanking heuristic[vi]

[vi] This figure refers to "dots" which are the same as "food pills". These two terms can be used interchangeably.

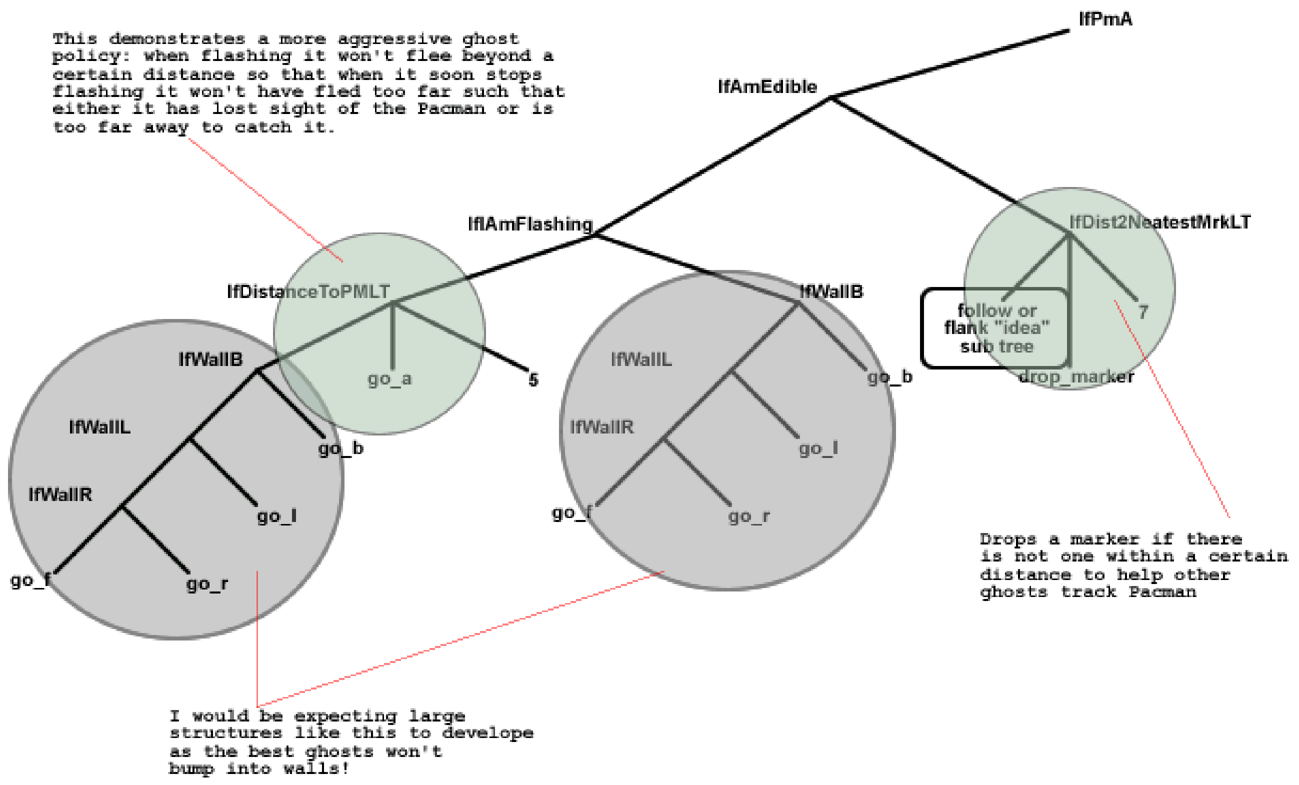

This demonstrates a more aggressive ghost policy: when flashing it won't flee beyond a certain distance so that when it soon stops flashing it won't have fled too far such that either it has lost sight of the Pacman or is too far away to catch it.
IfPmA
IfAmEdible
IfIAmFlashing
IfDist2NeatestMrkLT
IfDistanceToPMLT
IfWallB
go_a
5
IfWallB
follow or flank "idea" sub tree
7
drop_marker
IfWallL
go_b
IfWallR
go_l
go_f
go_r
IfWallL
IfWallR
go_b
go_l
go_f
go_r
Drops a marker if there is not one within a certain distance to help other ghosts track Pacman
I would be expecting large structures like this to develope as the best ghosts won't bump into walls!

Figure 15 – An advanced example strategy

Evaluating Strategy Performance

Now that the models have been explained, the final step is to show how the solutions found by both the GA and the GP methods would be evaluated.

The first point to note is that the game is noisy, and thus the evaluation function does not return a strategy's true value. This is because the strategy is evaluated in terms of the performance of the ghost who adopts this strategy. The same strategy used, with the same peer group of ghosts on the same maze against the same Pacman, could potentially produce a different performance on each round played. Thus noise is added to the fitness landscape

The first idea was to use a linear weighting of performance statistics such as:

w_1 * *number of appropriate moves made in chase* – w_2 * *number of times eaten* – w_3 * *number of suicidal moves made* + …

This however introduces a search and optimisation problem of its own because the strategy evaluation itself would need to be optimised to find the set of linear weights that would evoke the optimal strategy. Thus this method was abandoned.

The evaluation function decided upon is one that is designed to reward ghosts for being close to the Pacman at the end of a round. It heavily penalises ghosts for being eaten. Previous test runs had shown that this penalty needed to be severe to encourage fleeing behaviour to be learnt. Thus each ghost was evaluated separately using these parameters. It should be noticed that when a ghost dies, it is recreated as it was. It does not learn during a game as this would mean it learns during an evolutionary cycle and then can pass what's learnt onto its offspring which would imply a Lamarckian evolutionary model [3]. In an extension to this, each ghost has its fitness score scaled by the number of food pills that the Pacman has been able to eat. This is applied equally to all ghosts in the group. This last extension was used to help encourage searching as fitter ghosts will catch the Pacman sooner. It also helps to provide cross generation comparability. The evaluation function is shown below.

$$F = e^{(\text{Longest shortest path} - \text{Distance to Pacman via shortest path})/7.0} * 0.5^{\text{number of times eaten}}$$

$$\text{Fitness} = F * (\text{\#pills at start} - \text{\#pills at end}) / \text{\#pills at start}$$

The reader will note that "Fitness" has the distance placed on an exponential scale. This has been done on the basis of the initial investigations into GA which showed that selection pressure needed to be applied with such measures of distance to produce good results (see Figure 7). The dividing factor of 7.0 is particular to the maze over which the ghosts were being trained and is used to ensure that the selection pressure is not too great.

Noise in the Evaluation Function

As has previously been mentioned the evaluation function is noisy essentially introducing noise into the fitness landscape. Fogel and Michalewicz [1] make a good discussion on the effects of noise in search, more specifically why GA should be a versatile and robust solution in such environments and the various methods such as "niche and specification" and " diploidy and dominance" which have also been used to aid GA effectiveness in noisy environments. For discussion in greater depth of the latter two techniques see Goldberg [2].

In an attempt to "smooth out" the noise added to the fitness landscape, all gho sts are made to play five rounds during each generation, their fitness being the average of their scores for the five rounds played.

Results

In every game the ghosts play against a Pacman which works on a position evaluation model using hand coded values. The Pacman plays well, having been "tweaked" by hand over initial test runs. Because these runs involved watching the evolutionary process, "tweaking" the Pacman took some time and is not as optimal as an evolved solution would be. Some particular points that were noted were:

a) A Pacman who favoured eating dots above eating edible ghosts meant that the ghosts in general did not learn to flee the Pacman when they were edible or did so only at a very slow rate. Learning would only occur when a ghost was "accidentally" in the path of the Pacman; otherwise a ghost could simply chase the Pacman without fear of being eaten. Having a very aggressive Pacman solved this problem, but it should be noted that human players tend to use the period of edible ghosts to eat as many dots as possible rather than eat ghosts.

b) To avoid making mistakes such as running into a ghost which only became invincible in its last turn the Pacman had to view flashing ghosts as being undesirable. Such a strategy does not correlate very well with the way a human player might play. A human player would tend to be able to eat the majority of flashing ghosts based on how long they had been flashing. As such the Pacman strategy behaves slightly more naively than a human player in this specific respect.

c) A small modification to the position evaluation strategy allowed the Pacman to choose not to move, allowing him to hide from near by ghosts (recall that Pacman plays as nature). This is not allowed in the traditional game, but this makes the Pacman a more effective opponent, encouraging the ghosts to search more effectively.

The hand coding of the Pacman position evaluation and the time taken in doing so implies that co-evolution would be particularly useful here. This however was out of the scope of this project and was as such never an aim. Co-evolution will be discussed as an idea for future work.

Both methods use an initial population which is randomly generated.

Results for the Position Evaluation Model

The position evaluation model was the most successful out of the two models being compared. It can be seen that the ghosts learn how to play. They develop the ability to chase and flee when appropriate. They also develop the ability to make flanking manoeuvres, albeit naively.

Each evolution run took about one day and one night to complete, due to a population size of 500 being used. This relatively short running time meant that this model in particular was used to complete more trials, especially in an attempt to elicit more details about the effect of crossover rates on search. Another point to note is that the population size proved to be more than sufficient to provide enough diversity; this was not so in the GP model as will be discussed.

The results have been collected using

1. Just Roulette selection with
    a. No crossover with Jungian mutation rate
    b. Jungian crossover rate (60%) with Jungian mutation rate (3.33%)
    c. 100% crossover with Jungian mutation rate

    in order to make further investigations into crossover rates, given the inconclusive results from the initial investigations into GA.
2. Roulette selection and SSWR with strong and weak elitism. This compares the usefulness of a stochastic error reducing selection method and the effect of Jung's elitism in a *noisy* environment.

The results which will now be presented have been averaged over 3 evolutionary runs for each of the above points, and will be dealt with in the order given above.

Crossover Comparison Results

Figure 16 shows that the effect of crossover rates (using a constant level of mutation in all runs of 3.33%) has little to no effect on the rate of development of the population. With or without crossover, the rate of increase of games won is much the same, and the maximum win-proportion achieved also stays much the same independent of the rate of crossover. This correlates well with the initial investigations into GA, the end result being that this project has not found any empirical evidence to support the use of simple crossover as a search technique. Comparing average fitness of populations also reveals the same trend in that crossover doesn't affect rates of increase.

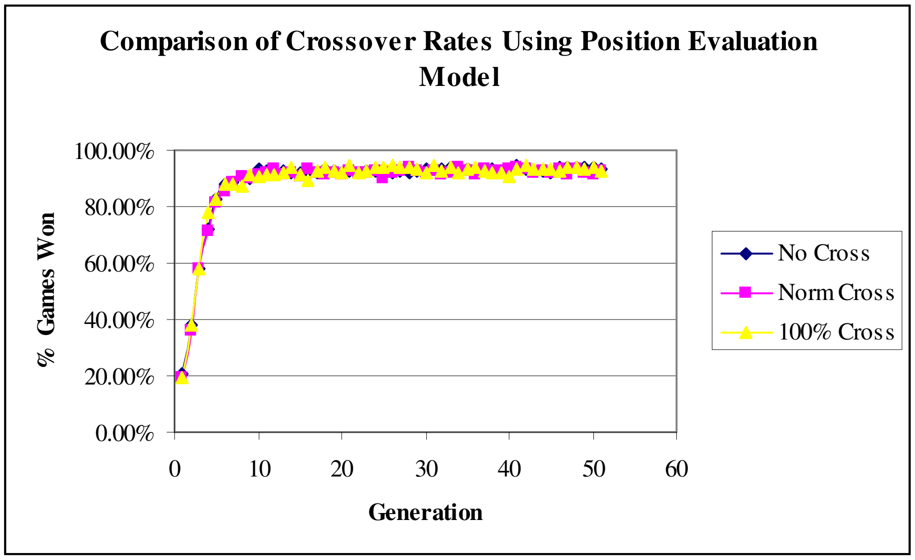


Figure 16 – Comparison of the Effect of Crossover Rates on the Position Evaluation Model

Another point to note from Figure 16 is that by generation ten the population has almost reached a steady level of proportion of wins. Most of the evolution appears to be occurring in the first ten generations. This is very fast rate of convergence. Given that the percentage of wins is very high this does not imply premature convergence but suggests quick convergence to a near optimal solution.

The fast rate of convergence does not necessarily mean that better solutions are being developed. It could mean that the average solution of the population is just being improved. To verify that useful evolution is taking place the fitness of solutions must be examined. Given that the effect of crossover has been negligible only one of the models listed under point one need be considered.

Figure 17 shows that the fitness of the best solutions per generation is increasing. This serves to verify that better solutions are indeed being found. We also see that the average fitness of the population is increasing in proportion to the best fitness. Average fitness is not growing at a rate that is faster than best fitness which indicates that even by generation fifty there is still much diversity in the population. Although only fifty generations are shown, runs made up to one hundred generations do not show further fitness increase.

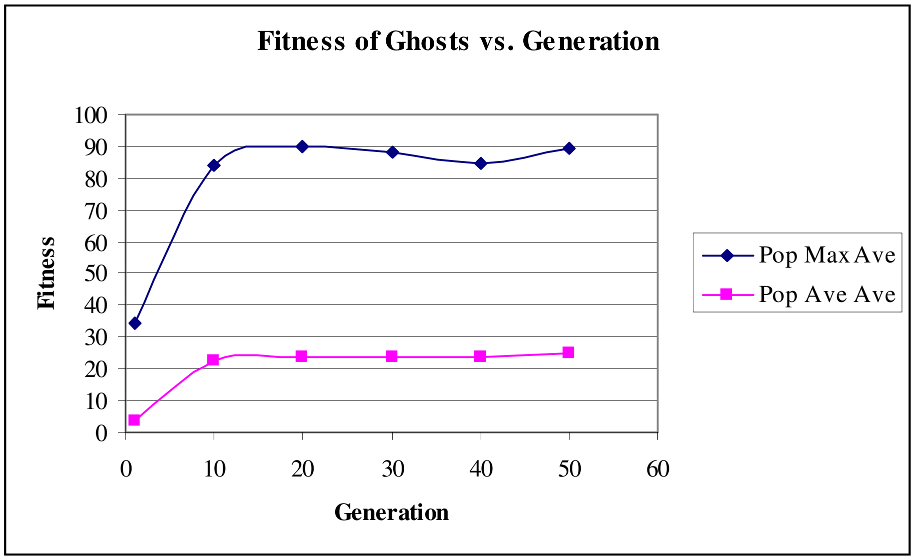


Figure 17 – Ghost Fitness per Generation in Position Evaluation Model Using Roulette Selection and No Crossover

It is also worth noting that the best group of four ghosts from each generation do not die more often than twice in any round. This indicates that the search and chase strategy of ghosts is increasing. In other runs when ghosts do not appear to have this initial knowledge, they are seen to learn it.

Figure 18 shows how the set of four fittest ghosts from the last generation in the evolutionary run discussed above have learnt to make flanks. The ghosts in grey represent the previous positions of the yellow and red ghost, the grey ghost furthest to the left being the yellow ghost. One can see how the red ghost (in its position denoted by its grey ‘shadow’) has learnt to make correct position evaluations: the squares around the yellow ghost are seen as being undesirable so it did not move north. Secondly due to the implicit communication being used the red ghost was able to evaluate the squares near the Pacman as being desirable and thus decide to move south, completing a flanking manoeuvre.

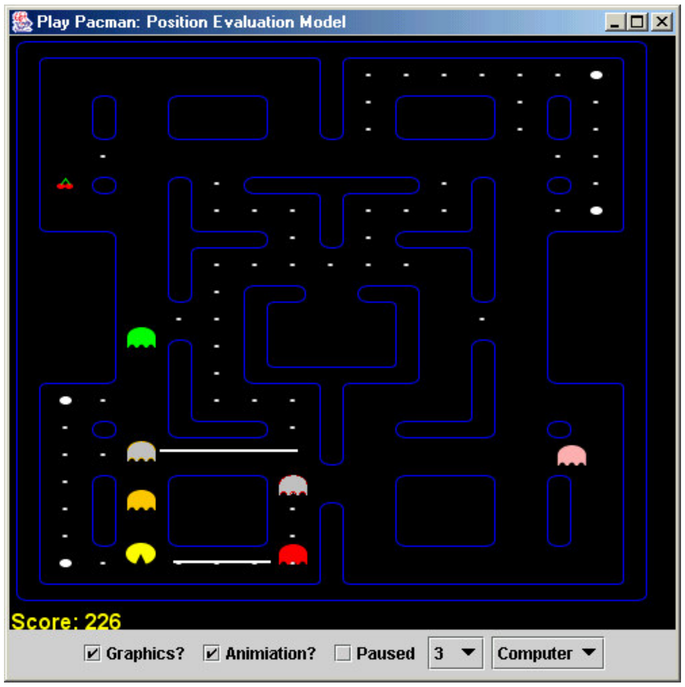


Figure 18 – Ghosts learn to flank

As a contrast, Figure 19 shows that three out of the set of four fittest ghosts from the first generation do not even know how to search out of their ‘home”, and in fact two ghosts are occupying the same square constantly.

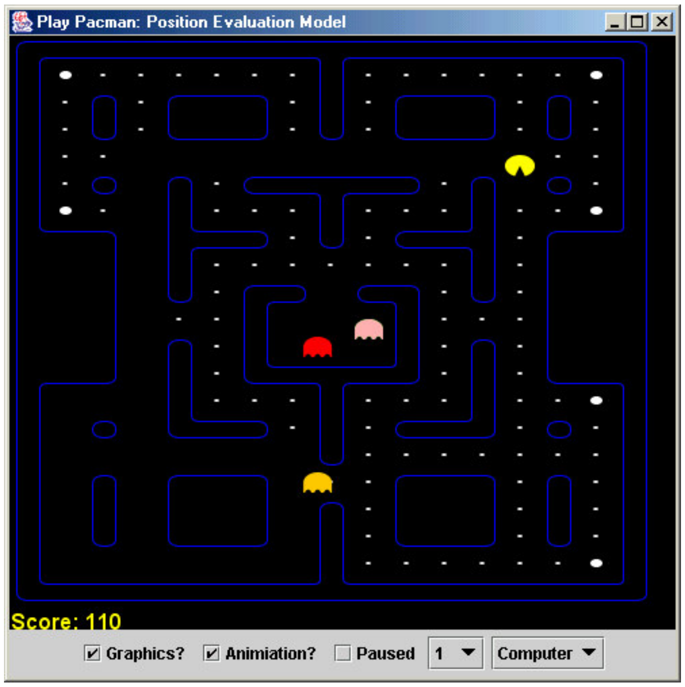


Figure 19 – The Best Ghosts from the Initial Population Perform Poorly

The lack of results to support evidence of a positive effect on search by the crossover operator was disappointing. It may be that such an operator is not appropriate for such a multi-parameter problem. A more promising recombination operator to investigate could be "majority logic crossover" [1], where parameters in a child chromosome are set to agree with the most frequent value from a set of n parents selected from the population in some manner (i.e. randomly as in Tournament selection, or using a fitness proportional or deterministic selection method). This will be discussed as an idea for future work.

Comparison of stochastic error reduction techniques results

Figure 20 shows a comparison between SSWR and Roulette selection using 1% and 20% elitism for each method over the first ten generations of each evolutionary run. Only the first ten generations are shown as the previous section has shown that most of the evolutionary work occurs within this period. The results are not as clear cut as one might have expected.

One would expect the stochastic error reducing techniques (SERT), of which SSWR is one, using the more extreme elitist value to have converged more slowly as they would be more likely to maintain less valuable solutions given the noise in the evaluation function which could lead to suboptimal evaluations of a solutions true worth. This seems to be verified using SSWR: when 1% elitism is used the solution is converged upon at a slightly faster rate than when 20% elitism is used. Indeed at about generation six, the later seems to make a "mistake" and the percentage of wins drops significantly, lending credence to the predictions.

The results of using Roulette selection are not as initially expected however and shed further light on this analysis. The higher rate of elitism with Roulette selection initially outperforms its peer using only 1% elitism, although this eventually finds what appears to be the better solution in terms of percentage-wins. It is possible that the use of Roulette, with it's inherit stochasti c error property, is compensating for the effect that is observed with higher elitism in the comparison of the two SSWR curves. Stochastic errors are perhaps allowing solutions evaluated to less than their actual worth due to noise to gain greater representation in the selection process.

It is interesting that Roulette with 20% elitism has been able to perform almost as well as SSWR with 1% elitism. This appears to indicate that it is not elitism that has a dampening effect on convergence in a noisy environment but SERT, which more closely approximate deterministic selection. This helps to explain why SSWR with 20% elitism performs poorly compared to SSWR with 1% elitism. If it is the case that it is not elitism which slows the convergence, as this only maintains a section of the most valuable solutions, the results would then appear to indicate that it is the SERT which introduce this dampening effect, as it would be effective globally, across an entire population (as selection is more deterministic-like and the evaluation function is trusted), and that this effect is only reinforced by the elitism. This explanation would serve to explain why SSWR with 1% elitism outperforms SSWR with 20% elitism, and also why Roulette with 20% elitism can compete with SSWR with 1% elitism.

These results are certainly interesting and shed light on the effect of noise on the effectiveness of various selection methods, and thus how choice of these evolutionary factors would be made dependent on the environment in which they will be used. In summary it would appear that elitism is not necessarily a bad choice for both noiseless and noisy environments. However, in noisy environments it appears to be the case that SERT is not necessarily a good strategy to assume.

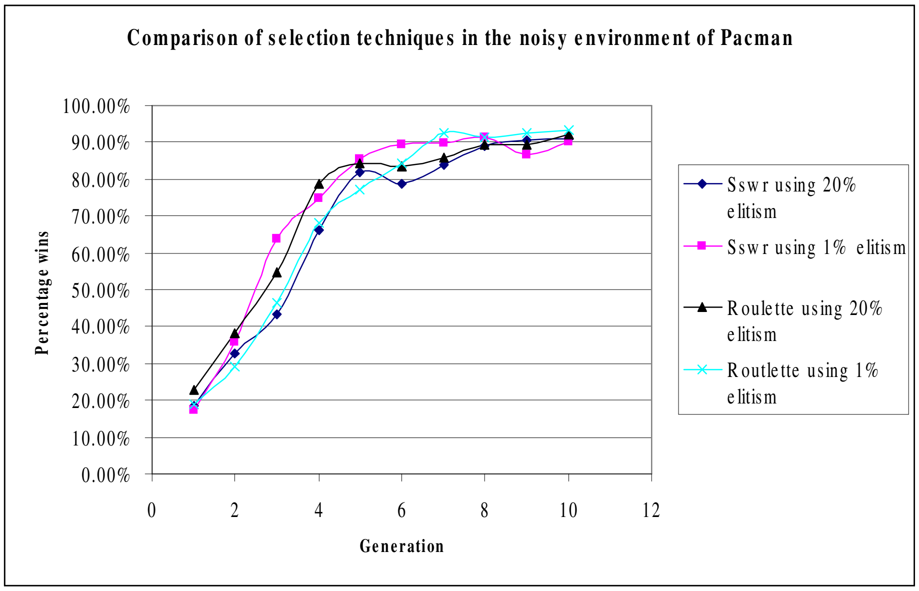


Figure 20 – A comparison of stochastic error reducing techniques and standard selection techniques in the noisy environment of Pacman

Results for the GP Model

The GP model proved to be the less successful of the two models being compared. The population size needed to provide enough initial diversity was 2000 or more individuals, which meant that the evolutions had to be run over an entire week. Thus there was significant time overhead in gaining results for this model. The large population sizes required also meant extending the maximum heap size available to Java Virtual Machine (see appendices for an explanation).

GP with Niching

The initial test population for this model was five hundred individuals. This proved far too small as by the twentieth generation all diversity would be lost. However, before the Java heap size was extended, a niching method was investigated. The method chosen was Cavicchio's preselection mechanism. It was chosen above fitness sharing and De Jung's crowding model (see Niching and Diversity) as it was the most natural to implement, especially given the coding used. The reason for this is that in GP it becomes very difficult to specify how "different" or how "similar" two solutions are. It would be very difficult to come up with some meaningful and accurate measure for this given the size of the function set used and the fact that individuals in the population are not all of equal size or structure. Thus Cavicchio preselection was the most natural to implement.

A run made using preselection with a population of 500 individuals was made. The results were disappointing. Although preselection did allow the population diversity to survive into later generations than the run without preselection (there was still some diversity in the late thirtieth to fortieth generations), it proved to be too constraining. Because trees could only replace their parents if they were fitter it led to a few super strings developing. Super strings are not necessarily ultra fit strings; they are just strings with fitness much higher relative to their peers in the population. Because super strings develop using preselection, the intermediate population tends to be dominated by them. This has the implication that although much diversity might be maintained, it cannot be used effectively as the less fit, diversity-providing, strings cannot compete in the evolutionary process which creates the intermediate population.

One should also note that because the evaluation function is noisy the GA cannot reliably replace parents with their offspring.

A section of the best string produced using the niching method is shown below:

```
(IF_PPIL_RIGHT
      (IF_FLASHING
            (IF_MARKER_LEFT
                  (IF_DOTS_BEHIND
                        (MOVE_RANDOM)
                        (MOVE_BEHIND)
                  )
                  (IF_WALL_LEFT
                        (IF_GHOST_AHEAD
                              (MOVE_BEHIND)
                              (MOVE_LEFT)
                        )
                        (IF_MARKER_RIGHT
                              (MOVE_AHEAD)
                              (IF_NUM_DOTS_RIGHT_LT
                                    (IF_MARKER_RIGHT
                                          (MOVE_LEAST_RECENT)
                                          (DROP_MARKER)
                                    )
                                    (IF_GHOST_RIGHT
                                          (MOVE_RIGHT)
                                          (MOVE_AHEAD)
                                    )
                                    (DISTANCE_TO_NEAREST_MARKER)
                              )))) … )
```

If there is a power pill to the right and the ghost is flashing and there is a marker to the left and dots behind move randomly. If there are no dots behind move behind

If there is a power pill to the right and the ghost is flashing and there is *no* marker to the left but there is a ghost ahead move behind otherwise left.

If there is a power pill to the right and the ghost is flashing, there's no marker and no wall to the left but there is a marker to the right, move ahead. If there is no marker to the right but the number of dots to the right is less then the distance to the nearest marker move to the least recent square visited …

As can clearly be seen, not only is this one portion of the tree very large and very complex, it also does not provide a sensible strategy: the reasoning is convoluted and erratic as can be read either from this portion of the tree or the annotations made beside each sub tree.

Due to these initial results a larger population was used with a 'kick start" as explained below.

GP with Large Populations

Given the disappointing results from the first two trials it was decided that when using the larger population size of 2000 ghosts, a small proportion, about 10%, of the initial population would be constructed not randomly but with simple hand written decision trees which were meant to act as a 'kick start". The hand coded trees are shown below in Figure 21 and Figure 22.

```
(IF_PACMAN_AHEAD
      (MOVE_AHEAD)
      (IF_PACMAN_BEHIND
            (MOVE_BEHIND)
            (IF_PACMAN_LEFT
                  (MOVE_LEFT)
                  (IF_PACMAN_RIGHT
                        (MOVE_RIGHT)
                        (MOVE_LEAST_RECENT)
                  )
            )
      )
)
```

A simple yet effective chase strategy with a poor search strategy as it does not take into account the position of other ghosts.

Figure 21 – Hand coded GP tree: An effective chase strategy

```
(IF_GHOST_AHEAD
      (IF_GHOST_LEFT
            (IF_GHOST_RIGHT
                  (IF_GHOST_BEHIND
                        (MOVE_LEAST_RECENT)
                        (MOVE_BEHIND)
                  )
                  (MOVE_RIGHT)
            )
            (MOVE_AHEAD)
      )
      (MOVE_LEFT)
)
```

A simple yet effective search strategy – ghosts will not make redundant searches. However they will not chase the Pacman.

Figure 22 – Hand coded GP tree: An effective search strategy

The third tree added consists of the first tree with "move least recent" replaced by the entire second tree. This would create the most basic search and chase strategy possible and ignores the idea that ghosts should run from the Pacman when they are edible. Using these three trees as a minor addition to a randomly generated population it was hoped that GP could build on these to create some strategies akin to the ones shown in Figure 14 and Figure 15. Having added hand coded trees, tree sizes were also restricted as large trees tended, in the previous two runs, to be either nonsensical or contain massive redundancies, or both. Thus any tree generated with more than twenty five nodes was given a "death penalty" (an extremely small fitness value) by the evaluation function.

The results using this third scheme were again disappointing. The GA did not produce strategies much more complex than those used to initially kick start the population. Some evidence of heuristic development can be seen; however by the time such trees are being developed, population diversity is becoming insufficient. This suggests the need to use an even larger population size.

Figure 23 shows the fittest solution tree from the last run with a population size of two thousand. The highlighted portion shows what appears to be the start of a reasonable heuristic.

```
(IF_PACMAN_AHEAD
       (MOVE_AHEAD)
       (IF_PACMAN_BEHIND
              (MOVE_BEHIND)
              (IF_PACMAN_LEFT
                     (MOVE_LEFT)
                     (IF_PACMAN_RIGHT
                            (MOVE_RIGHT)
                            (IF_WALL_LEFT
                                   (IF_DOTS_RIGHT
                                          (IF_NUM_DOTS_RIGHT_LT
                                                 (MOVE_LEAST_RECENT)
                                                 (MOVE_AHEAD)
                                                 (DISTANCE_TO_NEAREST_DOT_BEHIND)
                                          )
                                          (MOVE_LEAST_RECENT)
                                   )
                                   (IF_NUM_DOTS_BEHIND_LT
                                          (MOVE_AHEAD)
                                          (MOVE_RIGHT)
                                          (5)
                                   ))))))
```

A reasonably useful heuristic has been found.

Figure 23 – Best solution from GP model

The heuristic found is an attempt to make a promising search of the maze when the Pacman can't be seen by choosing to move ahead when there are not a sufficient number of dots behind. This can be considered a useful search strategy if it can be assumed that firstly there are more pills ahead and secondly that the Pacman is more likely to be in an area of the maze where more pills exist. Although this looks promising, it is not getting sufficiently close to the complex heuristics (see Figure 14 and Figure 15 for examples) that GP seemed to promise.

It was mentioned earlier that trees tended to contain redundancies. This occurs when reasoning takes the form of 'if object_0 not ahead then [… if object_0 ahead then <*redundant block*> ..] else … ", for example. A simple method to remove such redundancies would be to prune each tree before every evolutionary cycle. However, given the results for this method and the function/terminal set chosen, it was decided that the introduction of redundancy pruning would probably have little effect as larger populations could be obtained by increasing the Java heap size limit or giving trees a "death penalty" when they contain too many redundancies. It is also possible that redundant sub trees could also contain diverse elements and still be of use in recombination.

Having said that GP has not proved itself in this test, it can at least be seen that the population best and average fitness do improve as the population begins to converge to a solution. In fact, fitness levels equivalent to the first model are obtained, as shown in Figure 24. However, the solution converged to, as was shown above, tends to be one not very different (if at all in some cases) from the hand coded solutions added to the population in the first generation. This only serves to verify that the evolutionary process has settled on the most basic solution available to it, which means that even these simple hand coded solutions at least have some effect against the Pacman controller used. General population fitness measures only increase because as evolution proceeds the simple hand coded solutions become more numerous and so play in groups consisting mainly of themselves and insignificant variants of themselves rather than in groups of the randomly generated, much poorer trees; thus although fitness improves, this does not give merit to the GP and the strategies evolved.

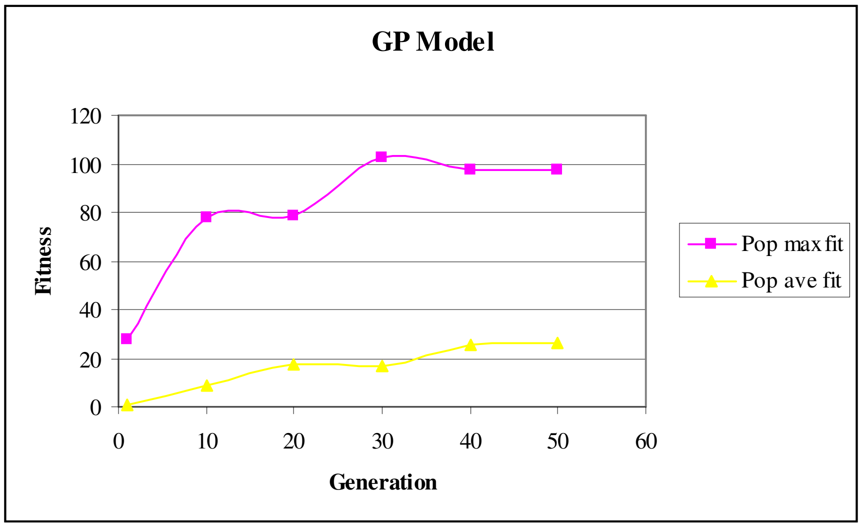


Figure 24 – Fitness growth in the GP model

Future Work and Conclusions

It is the author's belief that the project provides a good and thorough investigation into Evolutionary/Genetic Algorithms. It has shown that GA applied to fixed length strings are an effective tool in search and optimisation problems, showing robust and quick searching ability in a noisy environment. Some useful results have been obtained in both models, whether in support of evolutionary methods or not, and interesting avenues for further work and experiment have also been opened.

Conclusions for the GA model used reflect advice given in many textbooks: that low cardinality alphabets used over a fixed string representation perform well. The investigation into evolutionary variables has proved enlightening and provides a look at GA not found in the main sources used for this project [1, 2, 3]. The results obtained for crossover have raised more questions than they have answered, except to say that single point crossover used with a multi-parameter coding as was used here does not seem to contribute much to the algorithm's success. Conclusions for the GP model are somewhat inconclusive except to show limitations of the complexity of programs that this method can evolve.

The most important possible extension to the project would be to make a comparison between GA and a more traditional search technique like simulated annealing. Although this was not in the original project specification it was hoped that such a comparison could have been made. However this has fallen outside the time span available for the project given the amount already accomplished and the learning curve and development issues involved. There seems to be some discussion in the EA community about the effectiveness of EA against algorithms as simple as hill climbing [1, 3].

Co-evolution is also another possible avenue to investigate. By using co-evolution the test case, i.e. the Pacman controller, can get better and better "… evolving to specifically target weaknesses … [forcing] the population ... to keep discovering new ... strategies" [3].

Another worthy extension would be to tune the GA used in the position evaluation model; more specifically to investigate different forms of crossover, the most promising crossover method being 'Majority Logic", which has already been mentioned in "Crossover Comparison Results".

Investigations into the apparent failings of the GP method used should also be made. It is most likely that the function/terminal set used is too large. At every node in a tree there are sixty two possible node types. Given that tree size was limited to twenty five nodes, this gives a search space of size 5.912e+53! Allowing larger trees or even an open ended search space increases this size exponentially to infinity. This is a very strong indicator that not even GA-like methods can deal with such large search spaces. It is also the case, as the initial GA investigation indicates, that the cardinality of the alphabet being used is severely reducing the effectiveness of the search. This suggests the need for a combination of reducing the alphabet cardinality and using redundancy pruning to allow smaller trees to represent strategy-equivalent larger trees. Reducing the alphabet cardinality appears to require trivialising the problem domain as discussed in the section "GP Applied to Pacman". So how can we add the power of abstraction to the encoding without trivialising the problem? John Koza has developed two possible answers to complex domains which might need to build complex solutions built on simpler solutions; namely automatic function definition and hierarchical automatic function definition [9]. Such methods might allow the GP to develop more abstract function sets like 'ifPacmanVisible()" and 'followPacman()" though evolution, for example, which can then be used directly in decision trees. This is desirable because it does not mean trivialising the problem domain a priori by design, but through the evolutionary process. An alternative to GP using decision trees would be to use a GA to represent finite, fixed length trees (programs) [10]. This appears a promising avenue as the report has already seen that GA working on fixed, low-cardinality alphabet strings are indeed effective tools for search.

A final, less significant extension, as it does not so much concern GA but the model used for PE games, would be to implement a probabilistic model. This would take an almost identical form to the position evaluation model except that probability weights would have to be evolved [11].

Acknowledgements

□ *Dr Paul Goldberg*

For his constant project supervision and guidance.

□ *Andrew Boot and his project "Using Genetic Algorithms to develop simple predator hunting strategies" for Dept. of Computer Science at Warwick University 2000/2001*

For the use of his project as an introduction to GA and initial thoughts on the use of a position evaluation model.

□ *All reference books and articles*

Especially Fogel et al [1] for such revealing discussions on evolutionary computing and Goldberg [2] for an in depth look at more specifically genetic algorithms. For further articles and summaries of other material read please see the project web site available at http://www.jambojames.me.uk.

Appendix A: JavaDoc for Evolution Engine and Pacman

The Java Documentation for both the Evolution Engine and the Pacman game components can be found on the accompanying disk **#2**. The documentation has not been included here due to its length.

Both the packages are reusable and can be easily used in any program. The Evolution Engine is a non-visual tool. The Pacman game component is a Swing component which extends JPanel. It implements the game logic and relies on both the MazeGrid and PacmanMazeGrid components to supply the graphical drawing routines and a container for the maze state.

Documentation is in html format and can be viewed in any web browser. It is located on disk in the following directories and on the project website:

/SourceForEE/JavaDoc/index.html

Pacakages:

- uk.me.jambojames.evolutionengine

  Contains all classes used by the Evolution Engine as shown in "Figure 4 – The Evolution Engine design"

and

/MazeJavaSourceFiles/JavaDoc/index.html

Packages:

- uk.me.jambojames.pacman

  Contains classes used in implementing the PacmanGameComponent and game logic.
- uk.me.jambojames.pacman.pacmanmazegrid

  Contains the class used to implement the drawing logic used by the PacmanGameComponent's graphics display .
- uk.me.jambojames.simplemazegrid

  The simple maze grid definition (from which PacmanMazeGrid inherits) containing basic maze definitions, drawing logic, and methods used for statistics like shortest paths.

Appendix B: Run an Evolution or Game

Only the source code for the project is included on the disk **#1**. *Java 1.4 will be required.*

To run a game or evolutionary run the disk's root folder must be copied to another drive as there is not enough room on the disk provided for complied class files and evolutionary output. A *compilation script* ("compileAll.bat" for DOS or "compileAll.sh" for UNIX) is provided and must be used to compile the Java code for this project.

Once this is accomplished, for each model an evolutionary run can be executed or a game can be played against a selection of evolved and not-evolved ghosts (definitions provided in the *.ghosts files). Scripts for use with MS-DOS and UNIX are included for this purpose once the compilation stage has been completed.

To make any evolutionary run note that the evolutionary output data will be placed in /X/out/output.csv where X is either of the two directories GPGhosts or GridGhosts.
Use:

RunGpGhosts.bat or RunGpGhosts.sh

To run an evolution of the ghosts described in the GP model

RunGridGhosts.bat or RunGridGhosts.sh

To run an evolution of the ghosts described in the position evaluation model.

To play against some example ghosts use:

PlayGpGhosts.bat or PlayGpGhosts.sh

To play against ghosts using the GP model. The ghost definition files will be found in /GPGhosts/ and have extension *.gpghosts

PlayGridGhosts.bat or PlayGridGhosts.sh

To play against ghosts using the position evaluation model. Ghost definition files will be found in /GridGhosts/GhostFiles/ and have extension *.ghosts

Appendix C: Extending the Java Heap Size

This has been added as an appendix because some internet sources on the subject do not explain how to do this clearly or even in some cases correctly and because finding out how to do this can be time consuming when it need and should not be.

To extend the Java heap size correctly it is not the case that only the maximum size need be set. Instead the minimum heap size must be set to the initial amount of memory that the program will use. Then the maximum is set to the minimum plus a constant to provide a ceiling for the heap size. The command is shown below:

```
java –Xms???m –Xmx???m
```

The option –Xms???m sets the minimum heap size to ??? Megabytes. The option –Xmx???m sets the maximum heap size to ??? Megabytes.

References

[1] Michalewicz Zbigniew & Fogel B David. How to solve it: Modern Heuristics. Springer-Verlag Berlin Heidelberg. 2000.

[2] Goldberg E David. Genetic Algorithms in search, optimization and machine learning. Addison Wesley 1989.

[3] Mitchell Melanie. An introduction to genetic algorithms. MIT Press 1996.

[4] Russel Stuart & Norvig Peter. Artificial Intelligence: a modern approach. Prentice Hall 2003.

[5] Koza John R. Evolution of Subsumption Using Genetic Programming. Computer Science Dept. Stanford University. Available from the World Wide Web: http://www.geneticprograming.org (static link).

[6] Unknown authors. Genetic Programming of Hive Intelligence. 1998. Available from the World Wide Web: http://www.challenge.nm.org/Archive/98-99/finalreports/006/rec.htm (Accessed 9th October 2003).

[7]Michael LaLena. Teamwork In Genetic Programming. 1998. Available from the World Wide Web: http://www.lalena.com/ai/ant/parameters.shtml (Accessed 20th October 2003)

[8] Ehlis Tobin. Application of Genetic programming to the 'Snake Game". Available from the World Wide Web: http://www.gamedev.net/reference/articles/article1175.asp (Accessed 9th October 2003).

[9] Koza John R. Hierarchical Automatic Function Definition in Genetic Programming. Available from the World Wide Web: http://www.geneticprograming.org (static link).

[10] Cramer Nichael Lynn. A Representation for the Adaptive Generation of Simple Sequential Programs. Available from the World Wide Web: http://www.sover.net/~nichael/nlc-publications/icga85/index.html (Accessed 10th January 2004)

[11] René Vidal et al. Probabilistic Pursuit-Evasion Games: Theory, Implementation and Experimental Evaluation. IEE Transactions on Robotics and Automation, Vol. XX No. Y 2002

Please see project website at http://www.jambojames.me.uk *for a list of further article summaries on PE problems, GA and co-evolution.*