# Improving the Performance of a Computer-Controlled Player in a Maze Chase Game using Evolutionary Programming on a Finite-State Machine

Maximiliano Miranda and Federico Peinado

Departamento de Ingeniería del Software e Inteligencia Artificial,
Facultad de Informática, Universidad Complutense de Madrid
28040 Madrid, Spain
m.miranda@ucm.es, email@federicopeinado.com

**Abstract.** The continuous sophistication of video games represents a stimulating challenge for Artificial Intelligence researchers. As part of our work on improving the behaviour of military units in Real-Time Strategy Games we are testing different techniques and methodologies for computer-controlled players. In this paper Evolutionary Programming is explored, establishing a first approach to develop an automatic controller for the classic maze chase game Ms. Pac-Man. Several combinations of different operators for selection, crossover, mutation and replacement are studied, creating an algorithm that changes the variables of a simple finite-state machine representing the behaviour of the player's avatar. After the initial training, we evaluate the results obtained by all these combinations, identifying best choices and discussing the performance improvement that can be obtained with similar techniques in complex games.

**Keywords.** Video Game Development, Player Simulation, Artificial Intelligence, Machine Learning, Evolutionary Computing, Genetic Algorithms

## 1 Introduction

Video games are constantly experiencing improvements in graphics, interfaces and programming techniques. It is one of the most challenging and interesting field of application for Artificial Intelligence (AI), considering it as a large set of "toy worlds" to explore and play with. Recently, we have started working on how to improve the behaviour of military units in Real-Time Strategy (RTS) games. In this context, different techniques and methodologies for computer-controlled players as part of that research project are being tested. In this genre it is very common to implement the behaviour of a military unit as a Finite-State Machine (FSM). Some of these machines are very complex, having multiple parameters that should be adjusted by experimentation (playtesting) and by experienced game designers. We have decided to choose a simple game to start trying different techniques and methodologies (as it is the case of

Evolutionary Programming). If significant results are found, we could study their applicability for improving the more complex FSMs of RTS games.

For this paper we have developed an automatic controller for a version of the game Ms. Pac-Man implemented in a software platform called "Ms. Pac-Man vs. Ghosts". The behaviour of the protagonist (Ms. Pac-Man) in the game is implemented by a FSM with some variable values that act as a threshold to control the state transitions. A genetic algorithm is used to find the best values for the performance of this FSM, aiming to probe the utility of this approach improving the game score.

The structure of this paper is as follows: Section 2 is focused on the definitions of the concepts and the background needed to understand the rest of the paper. Section 3 presents our computer-controlled player for the Ms. Pac-Man game. Section 4 describes how we apply evolutionary computation to improve the performance (score) of the player and Section 5 shows and discusses the results of the experiments. Finally, in Section 6 we present our conclusions, foreseen the next steps of this project.

## 2 Related Work

Recently, we have started working on how to improve the behaviour of military units in RTS games with two main objectives: participating in international challenges of AI applied to video games, such as the Student StarCraft AI Tournament [1] and the AIIDE StarCraft AI Competition [2], and improving the multi-level FSMs used in Mutant Meat City, a RTS-style video game created in 2014 as an academic project.

Before applying evolutionary programming to complex games, we are performing experiments using more simple games, such as Pac-Man. Pac-Man is a popular arcade created by Toru Iwatani and Shigeo Funaki in 1980. Produced by Namco, it has been consider as an icon since its launch, not only for the videogames industry, but for the twentieth century popular culture [3]. Game mechanics consist on controlling Pac-Man (a small yellow character) who is eating pills (white dots) in a maze, while avoiding to be chased by four ghosts that can kill him (decreasing his "number of lifes"). There are also randomly-located fruits in the maze that add extra points to the score when eaten. The game is over when Pac-Man loses three lifes. There are also special pills, bigger than the normal ones, which make the ghost to be "edible" during a short period of time. The punctuation in the score grows exponentially after each ghost been eaten during this period.

Ms. Pac-Man is the next version of the game, produced in 1982 by Midway Manufacturing Corporation, distributors of the original version of Pac-Man in the USA. This version is slightly faster than the original game and, in contrast with the first one, the ghosts do not have a deterministic behaviour, their path through the maze is not predefined [4]. This makes the game more difficult, being much more challenging the creation of strategies to avoid being killed.

Over the years there have been several proposals in the academy in relation to using AI for maze chase games as Pac-Man, both for controlling the protagonist or the antagonists the game. Ms. Pac-Man vs. Ghosts League was a competition for developing completely automated controllers for Ms. Pac-Man with the usual goal of optimiz-

ing the score. For supporting this competition, a Java framework "Pac-Man vs. Ghosts" was created for implementing the game and making it easy to extend the classes for controlling the characters.

With respect to Evolutionary Computing, genetic algorithms were originally developed by Cramer [5] and popularized by Koza [6] among others. This paradigm has become a large field of study, being widely-used in AI challenges and for optimizing the behaviour of intelligent automata. These algorithms could be useful for optimizing controllers in video games, even in sophisticated titles using autonomous agents [7]; but for our purposes, a controlled and limited scenario as the simple mazes of Ms. Pac-Man is perfect to test the methodology for evaluating the performance of a computer-controlled player.

## 3 A Computer-Controlled Player for the Maze Chase Game

For this work, we have designed a simple controller for Ms. Pac-Man based on the *StarterPacman* class of the framework "Ms. Pac-Man vs. Ghosts". The controller implemented in this class is one of the simplest of the framework and it has been changed to transform it in a simple FSM with just three states:

- *Pilling*: Ms. Pac-Man goes to the closest pill in order to eat it. In case there were several pills at the same distance, it will follow a preference order according to the direction toward these and clockwise starting from the top.
- *Ghosting*: Ms. Pac-Man goes to the closest ghost in "edible" state.
- *Runaway*: Ms. Pac-Man runs away from the closest ghost.

In the implementation of this FSM we use four numerical variables that later on will compose the chromosome of the individuals of the population that the genetic algorithm will be using:

- *Edible_Ghost_Min_Dist*: The minimum distance that an "edible" ghost should be in order to start chasing him.
- *Non_Edible_Ghost_Min_Dist*: The minimum distance a ghost should be to start running away from him.
- *Min_Edible_Time*: The minimum time that make sense to be chasing an "edible" ghost.
- *Min_Pills_Amount*: The minimum number of pills that should stay in the level to start going toward them proactively instead of hiding from or eating ghosts.

Using these variables, the FSM has these transition rules:

- If the closest ghost is non-edible, its distance is inferior to *Non_Edible_Ghost_Min_Dist* and the number of pills near Ms. Pac-Man is lower than *Min_Pills_Amount*, the state changes to *Runaway*.

- If the closest ghost is edible, its distance is inferior to *Edible_Ghost_Min_Dist* and the number of pills near Ms. Pac-Man is lower than *Min_Pills_Amount*, the state changes to *Ghosting*.
- In other case, the state changes to *Pilling*.

## 4 Evolutionary Optimization of the Computer-Controlled Player

After the Ms. Pac-Man automatic player controller is explained, we perform the study to test if genetic algorithms can improve the FSM in terms of performance in the game.

### 4.1 Fitness Function

Genetic algorithms require an fitness (or evaluation) function to assign a punctuation to each "chromosome" of a population [8]. In this case the punctuation of the game itself will be used, so the game is executed with the parameters generated by the algorithm and average values are calculated after a constant number of game sessions played by a phenotype (set in 10). The average score from a set of played games acts as the fitness function for our algorithm.

### 4.2 Genetic Algorithm

When creating the genetic algorithm it has been implemented a codification based on floating point genes for the chromosome of the individuals. These genes are real numbers that take values between 0 and 1. The value of the genes are multiplied by 100 in order to evaluate the results. Indeed the FSM of Ms. Pac-Man controller needs values between 0 and 100. So on, every individual of the population represents a Ms. Pac-Man controller.

We have implemented two selection operators, six crossovers, two mutations and four substitutions (also called regrouping), in order to perform different tests and determine which combination of operators get the best results.

**Selection.** These are our two types of selection:

- *Selection by Ranking*: Individuals are ordered in a list according to their fitness. The probability for an individual to be chosen for the crossover is higher as higher is its average score.
- *Selection by Tournament*: *N* individuals of the population are selected randomly. Among these individuals the one with the better fitness value is selected.

**Crossover.** These are our six types of crossover:

- *One-Point-based Crossover*: The parental chromosomes genes are interchanged from a given gene position.

- *Multi-Point-based Crossover*: The parental chromosomes genes are interchanged from two given positions.
- *Uniform Crossover*: Two progenitors take part and two new descendants are created. A binary mask determines the division of the genes which are going to be crossed.
- *Plain Crossover*: *N* descendants are generated. The value of the gene of the descendant in the position *i* is chosen randomly in a range defined by the genes of the progenitors that are located in the same position.
- *Combined Crossover*: This is a generalization of the plain one, called BLX-alpha. *N* descendants are generated. The value of the gene of the position *i* in the descendant is chosen randomly from an interval.
- *Arithmetic Crossover*: Two new descendants are generated according to an arithmetic operation. The value of the gene *i* in the descendant X is the result of the operation (being A and B the progenitors and $A_i$ the value of the gene i of the chromosome A):

  $$X_i = r * A_i + (1 - r) * B_i$$

  And the value of the gene *i* in the descendant Y is the result of the operation:

  $$Y_i = r * B_i + (1 - r) * A_i$$

  *r* represents a variable real number, and for this experiment is set to 0,4.

**Mutation.** These are our two types of mutations:

- *Uniform Mutation*: A gene is randomly selected and it mutates. The value of the gene is replaced by another randomly generated.
- *Mutation by Interchange*: Two genes are selected and they interchanged positions.

**Substitution.** These are our four types of substitution:

- *Substitution of the Worse*: The descendants replace the individuals with worse fitness from all the population.
- *Random Substitution*: The individuals that are going to be substituted are randomly chosen.
- *Substitution by Tournament*: Groups of *N* individuals are selected and the worst of each group is replaced by a descendant.
- *Generational Substitution*: The descendants replace their own parents.

At the beginning, the population is initialized with a certain number of individuals (100 by default), all of them created with random genes. Each one is evaluated before it is added to the population structure (a tree structure is used to maintain the population ordered by the fitness value of each individual). Then, the minimum, maximum and average fitness of this population is calculated and the next generation is produced. This process is repeated several times (500 generations of individuals are created by default).

## 5 Results and Discussion

As it has been mentioned, using these operators of selection, crossover, mutation and substitution of individuals, we have been able to test different combinations of operators obtaining the results shown below. Instead of testing all the possible combinations (96 different experiments) and studying the interactions between operators one by one, as a first exploration of the problem we have taken a different approach. We have created a pipeline of "filters" for the operators, using heuristics based on principles of Game Design, so only operators offering the best performing results in their category are selected and the rest are discarded for the remaining experiments.

The graphics which are displayed below represent the results of the experiments. The X axis represents the number of the generation produced and the Y axis the values obtained in this generation.

### 5.1 Selection of the Substitution Operator

These operators are set: *selection-ranking, crossover-uniform*, *mutation-uniform*; and the different substitution operators are tested. See Fig. 1 and Fig. 2.

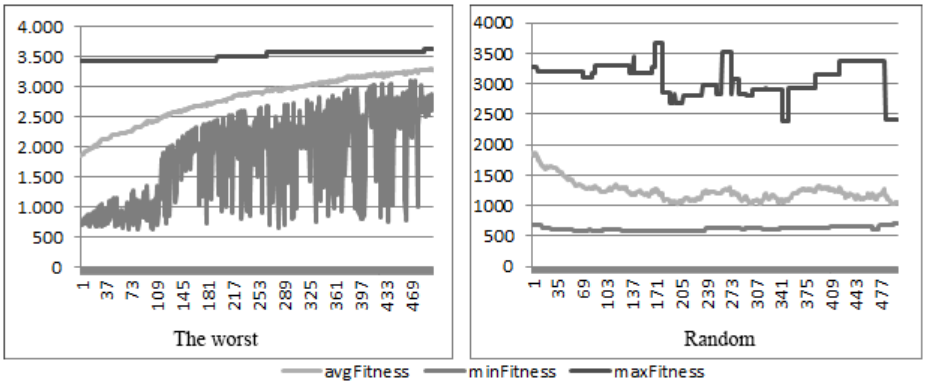


**Fig. 1.** *The Worst* and *Random* substitution

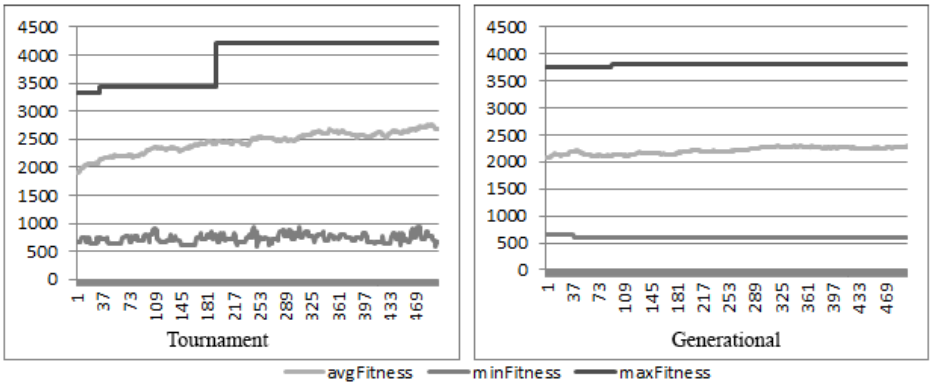


**Fig. 2.** *Tournament* and *Generational* substitution

As can be seen in the results, the substitution operator of the Worst was the one with better results in this algorithm. The random one is a little inconsistent, with big variations and making worse the average fitness (it is even worst: it discarded several times the best individual). The substitution by Tournament seems to work quite well but in other experiments, not shown in these graphs, it never improves the best individual. Finally, the generational replacement just does not improve anything. Therefore, the Substitution of the Worst operator is chosen as the best operator in its category.

### 5.2 Selection of the Mutation Operator

The next operators are set: selection-Ranking, crossover-Uniform, substitution-of the Worse (established in the previous point). Then, the different mutation operators are tested. The combination with the uniform mutation has already been tested in the previous step so we only have to test mutation by interchange. See Fig. 3.

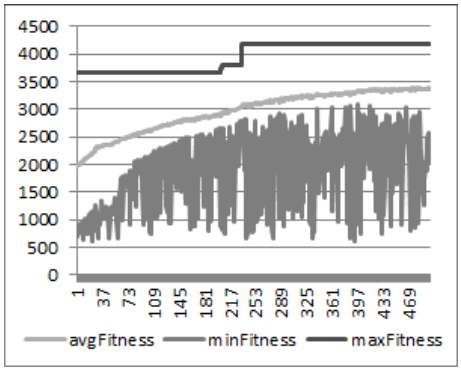


**Fig. 3.** Mutation by interchange

This method of mutation slightly improves the Uniform mutation operator, both in the average and maximum fitness where it gets some important jumps. Therefore we chose this method.

### 5.3 Selection of the Crossover Operator

After using the substitution operator (the Worst) and the mutation operator (by interchange), it is time to test the crossover operators (except the uniform one because it has been tested in first step of this pipeline). See Fig. 4, Fig. 5 and Fig. 6.

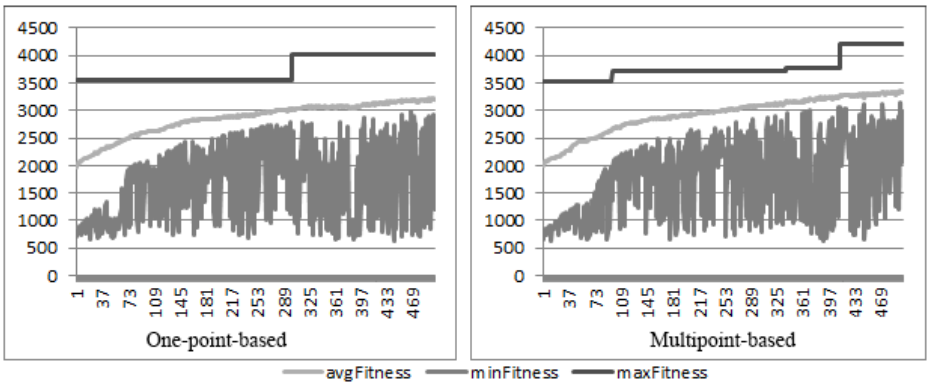


**Fig. 4.** *One-point-based* and *Multipoint-based* Crossover

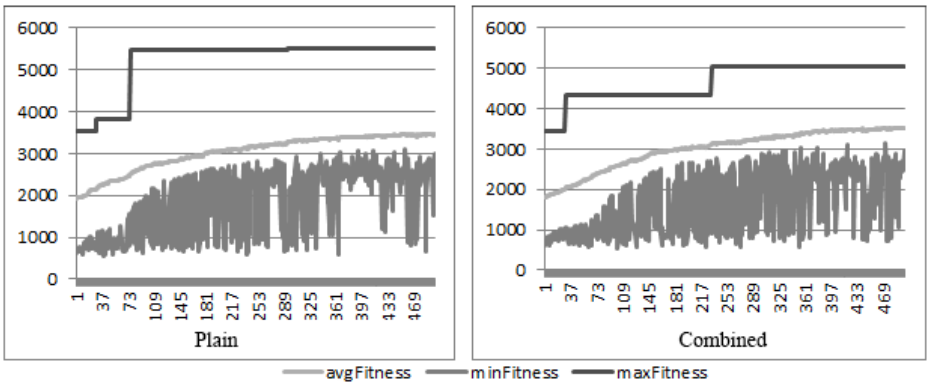


**Fig. 5.** *Plain* and *Combined* Crossover

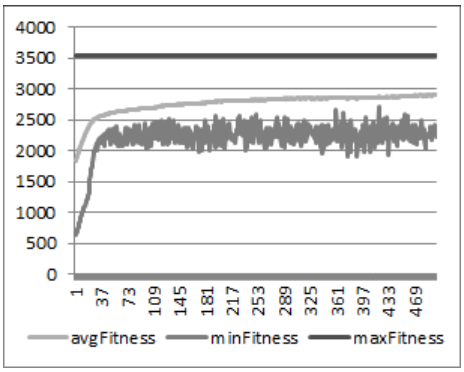


**Fig. 6.** *Arithmetic* Crossover

The One-Point-based operator get better results than the Uniform sometimes, but others it does not improve. The Multi-Point-based improves the results much more. The plain crossover produces few hops (improvements) in the best individual but these hops are very large, more than in the Multi-Point. The combined crossover also produces good results but, again, it does not produce as good improvements as the plain. Finally the arithmetic crossover produced a great improvement in the worse individuals (and thus the average fitness), however it does not improve the best. Therefore the Plain crossover is selected for the remaining experiments.

### 5.4 Choice of the Selection Operator

Once the operators of Substitution (the Worst), mutation (by Interchange) and crossover (plane) are set, it is the turn of testing the selection operators, in this case, the selection by tournament. See Fig. 7.

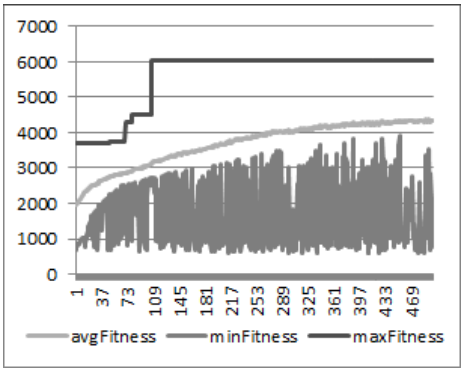


**Fig. 7.** Selection by *Tournament*

As can be seen in the graph, the selection method by tournament can produce large jumps in the best individual, in the most part of the experiments, it exceeds both max-

imum and average fitness to the selection by ranking method, consequently this selection method is selected and at this point we have selected all the operators for the genetic algorithm and it produces this individual with the chromosome with the maximum fitness: (120, 14, 324, 75). Let us remember that these values represent the variables used in the FSM of our controller for Ms. Pac-Man.

The operators of the genetic algorithm implemented selected by the results of these experiments are: *Selection by Tournament*, *Plane Crossover*, *Mutation by Interchange* and *Substitution of the Worst*.

## 6 Conclusions

The combination of operators selected after this particular game designer-style experimentation produce results that are far from the more than 20.000 points reached in the scores of official competitions of Ms. Pac-Man vs. Ghosts. But results seems reasonable considering the simplicity of the FSM developed here and the non-exhaustive methodology followed. The goal of this research was to double check that a genetic algorithm can improve significantly a FSM, even if it is a simple one and it is used as a practical testbed for game design. We could affirm that average scores of a population of FSM can be improved in more than 100%, and that the better ones also receive an improvement of approximately 60%.

Taking into account the simplicity of the implemented controller, it seems reasonable that a more elaborated one, for instance a multi-level FSM or a subsystem based on fuzzy rules for state transition, can be improved with this evolutionary approach.

Now we plan to repeat similar experiments with other combinations of operators, using a more rigorous approach, at the same time we add variations in some of the operators (for instance, changing the points in the crossing methods or modifying the function in the arithmetic crossing). Of course, our roadmap includes increasing the complexity of the FSM and starting to explore a simple strategy game.

## References

1. Student StarCraft AI Tournament (SSCAIT) http://www.sscaitournament.com/
2. AIIDE StarCraft AI Competition http://webdocs.cs.ualberta.ca/~cdavid/starcraftaicomp/index.shtml
3. Goldberg, H.: All your base are belong to us: How fifty years of videogames conquered pop culture. Three Rivers Press (2011)
4. Kent, S.L.: The ultimate history of video games: From Pong to Pokemon and beyond… The story behind the craze that touched our lives and changed the world. pp. 172-173. Prima Pub (2001)
5. Cramer, N.: A representation for the adaptive generation of simple sequential programs. International Conference on Genetic Algorithms and their Applications. Carnegie-Mellon University, July 24- 26 (1985)
6. Koza, J.: Genetic programming: on the programming of computers by means of natural selection. MA: The MIT press, Cambridge (1992)

7. Mads, H.: Autonomous agents: Introduction. (2010). Retrieved January 19, 2013 from http://www.cs.tcd.ie/Mads.Haahr/CS7056/notes/001.pdf
8. Melanie, M.: An Introduction to genetic algorithms (Complex adaptive systems). pp. 7-8. A Bradford Book (1998)