Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games (CIG 2007)

Evolving Pac-Man Players: Can We Learn from Raw Input?

Marcus Gallagher and Mark Ledwich
School of Information Technology and Electrical Engineering
University of Queensland, 4072. Australia
marcusg@itee.uq.edu.au, mark_ledwich@yahoo.com

***Abstract*— Pac-Man (and variant) computer games have received some recent attention in Artificial Intelligence research. One reason is that the game provides a platform that is both simple enough to conduct experimental research and complex enough to require non-trivial strategies for successful game-play. This paper describes an approach to developing Pac-Man playing agents that learn game-play based on minimal on-screen information. The agents are based on evolving neural network controllers using a simple evolutionary algorithm. The results show that neuroevolution is able to produce agents that display novice playing ability, with a minimal amount of on-screen information, no knowledge of the rules of the game and a minimally informative fitness function. The limitations of the approach are also discussed, together with possible directions for extending the work towards producing better Pac-Man playing agents.**

Keywords: Evolutionary Algorithm, Multi-layer Perceptron, Pac-Man, Neuroevolution, Real-time Computer Games.

I. Introduction

Pac-Man is one of the oldest and most popular computer games of all time. Following its release as an arcade game in the 1981, Pac-Man and variants of the game have proliferated from arcades to numerous personal computer platforms, consoles and hand-held devices. Although Pac-Man is simplistic in many aspects when compared with modern computer games, it remains a well-known and popular game.

In recent years, Pac-Man style games have received some attention in Computational Intelligence research. The main reason is that the game provides a sufficiently rich and useful platform for developing CI techniques in computer games. On one hand, Pac-Man is simple enough to permit reasonable understanding of its characteristics, requires relatively modest computational requirements and has a small code size. On the other hand, game-play based on "intelligent" strategies, planning and priority management is possible, as opposed to many other simple real-time computer games where success is based largely on speed and reaction-time. The predator-prey nature of Pac-Man provides significant challenges for using CI techniques to create intelligent agents in game-play.

This paper describes an approach to evolving agents that learn to play Pac-Man (i.e. assuming the role of the human controlled character). The agents are implemented as neural network (multi-layer perceptron) controllers. The main aim of the work described below is to explore the feasibility of this approach, when the neural networks are presented with minimal raw information about the state of the game. Section II describes key aspects of the Pac-Man game and variants. In Section III previous research in Pac-Man and neuroevolution in arcade games is reviewed. The approach taken in this paper is detailed in Section IV, while Section V presents the results of experiments. Section VI provides a discussion and summary of the work.

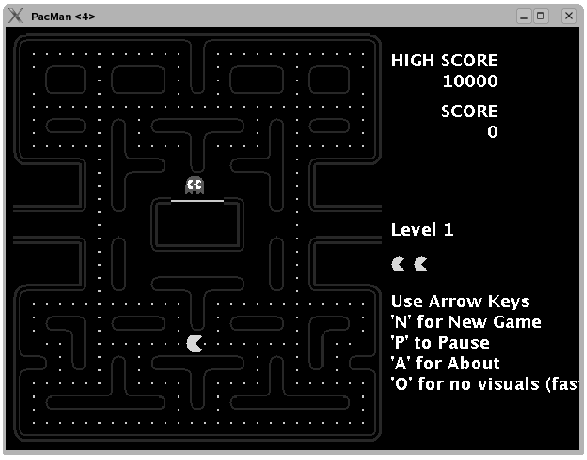

Fig. 1. The starting position of the Pac-Man game, showing the maze structure, Pac-Man (lower-center), power pills (large dots), dots (small dots) and ghosts (center).

II. The Pac-Man Game and Variants

Pac-Man is a simple predator-prey style game, where the human player maneuvers an agent (i.e. Pac-Man) through a maze. A screenshot at the start of a game (the version used in our research) is shown in Figure 1. The aim of the game is to score points, by eating dots initially distributed throughout the maze while attempting to avoid four "ghost" characters. If Pac-Man collides with a ghost, he loses one of his three lives and play resumes with the ghosts reassigned to their initial starting location (the "ghost cage" in the center of the maze). Four "power pills" are initially positioned near each corner of a maze: when Pac-Man eats a power pill he is able to turn the tables and eat the ghosts for a few seconds. Bonus "fruit" objects wander through the maze at random and can be eaten for extra points. The game ends when Pac-Man has lost all of his lives.

In the original game there are no elements of randomness - each ghost makes a deterministic decision about which direction to move at a given time-step. This makes it possible to devise effective game-play strategies based on learning patterns (following fixed paths) in the game [1]. Nevertheless, this determinism is not evident to the typical human player

since the ghosts movement is dependent on the position and movement of the Pac-Man agent. As long as the Pac-Man agent displays variety in play, the game dynamics will unfold differently. This means that typical human players develop strategies for playing the game based on task prioritization, planning and risk assessment.

While the basic behaviors of the ghost agents can be described, the precise (programmed) strategies in the original arcade game appear to be unknown (short of translating machine code back into assembly language for the Z80 microprocessor and reverse-engineering!). This highlights a more general issue for AI research on Pac-Man - different implementations of the game will have certain specific differences. In this paper we are mainly concerned with the main (non-implementation-specific) aspects of Pac-Man. Nevertheless, implementation-specific issues are important, e.g. in trying to compare different AI techniques for playing Pac-Man. Some examples of these issues are discussed further below.

III. Learning in Pac-Man and Related Previous Research

Several previous studies have used Pac-Man and variant games. Koza [2] and Rosca [3] use Pac-Man as an example problem domain to study the effectiveness of genetic programming for task prioritization. Their approach relies on a set of predefined control primitives for perception, action and program control (e.g., advance the agent on the shortest path to the nearest uneaten power pill). The programs produced represent procedures that solve mazes of a given structure, resulting in a sequence of primitives that are followed.

Kalyanpur and Simon [4] use a genetic algorithm to try to improve the strategy of the ghosts in a Pac-Man-like game. Here the solution produced is also a list of directions to be traversed. A neural network is used to determine suitable crossover and mutation rates from experimental data. Finally, De Bonet and Stauffer [5] describe a project using reinforcement learning to develop strategies simultaneously for Pac-Man and the ghosts, by starting with a small, simple maze structure and gradually adding complexity.

Gallagher and Ryan [6] used a simple finite-state machine model to control the Pac-Man agent, with a set of rules governing movement based on the "turn type" at Pac-Man's current location (e.g corridor, T-junction). The rules contained weight parameters which were evolved from game-play using the Population-Based Incremental Learning (PBIL) algorithm [7]. This approach was able to achieve some degree of learning, however the representation used appeared to have a number of shortcomings.

Recently, Lucas [8] proposed evolving neural networks as move evaluators in a Ms. Pac-Man implementation (Ms. Pac-Man was the sequel to the original Pac-Man game). Lucas focuses on Ms. Pac-Man because it is known that the ghosts in this game behave in a pseudo-random fashion, thus eliminating the possibility of developing path-following patterns to play the game effectively and presumably making the game harder and leading to more interesting game-play. The neural networks evolved utilize a handcrafted input feature vector consisting of shortest path distances from the current location to each ghost, the nearest power pill and the nearest maze junction. A score is produced for each possible next location given Pac-Man's current location. Evolution strategies were used to evolve connection weights in networks of fixed topology. The results demonstrate that the networks were able to learn reasonably successful game-play as well as highlighting some of the key issues of the task (such as the impact of a noisy fitness function providing coarse information on performance.

Evolving neural networks (aka neuroevolution) has emerged as a popular technique for agent learning in games [9]. Perhaps the most well known example is the work of Chellipilla and Fogel in the Anaconda/Blondie24 checkers-playing agent [10], [11]. A real-time version of the Neuroevolution of Augmenting Topologies [12] ((rt)NEAT) method has been developed and applied to a real-time game called Neuroevolving robotic operatives (NERO) [13], [14]. The method evolves a team of neural networks that learn to play a combat game in a 3D environment. rtNERO is based on a powerful and sophisticated type of neuroevolution where network topology is evolved together with weight values, starting with simple networks that gradually develop more complex architectures. Carefully designed evolutionary operators work well with the network model and input representation of the game.

In this paper, we take an approach inspired by this previous work utilizing neuroevolution in games. The work is similar to that of Lucas [8] but differs in several important respects. Primarily, we investigate the ability of neuroevolution to learn to play Pac-Man based on "raw" input from the current game state. Lucas speculates as to the feasibility of using on-screen input for learning Pac-Man: the results presented below provide a partial answer to this question. Our agent input is based on limited on-screen information and does not utilize computations or information that a human could not feasibly calculate or attend to online during game play (e.g shortest path calculations). The methodology is described in detail in the following section.

IV. Approach

A. Game Version Details

For the experiments we used a freely available Java-based implementation of Pac-Man developed by Chow [15]. This game implementation was also used in [6]. Chow's Pac-Man implementation follows the details of the original game reasonably well. Note however that in this version of the game, the behavior of the ghosts is *not* deterministic. In this respect the game is similar to the implementation of Lucas [8], although the manner in which randomness is used is almost certainly different. Based on a comparison of the source-code from [15] and the description in [8], there appear to be some differences in the team behavior of the four ghosts (e.g. Chow's implementation discourages the ghosts

from crowding together during play, while this phenomenon is noted by Lucas in the discussion of his results).

The code for the Pac-Man game used was modified to replace the keyboard input with output from the agents/neural network controllers. The networks are evolved in response to performance attained during game-play (see below). Since the game runs at an appropriate speed for humans to play, some effort was made to optimize the game speed for neuroevolution. Sound was disabled, and a feature added to disable the majority of graphical updates to the screen. These changes improved the game runtime. Unfortunately, the experimental simulations reported below still took a significant amount of time, due to the large number of games played when evolving a population of agents. On Pentium 4-based PCs, the experiments described in Section V each took between 1 and 4 weeks of CPU time (multiple game instances were run simultaneously on a single computer). The game also occasionally crashed for undetermined reasons. In general all experiments presented below were run for as much time as practically possible.

For the majority of experiments in this paper, the game was simplified by removing three of the four ghosts, all power pills and fruit. Consequently, the "first" (red) ghost moves by advancing towards Pac-Man by the shortest path 90% of the time, and 10% of the time will choose the second-best path to Pac-Man. If an internal game option ("InsaneAI") is set, the single ghost will become deterministic and always selects the shortest path to Pac-Man.

The maze layout is the same as in the original game. While it is commonly reported that this maze contains 240 dots [1], [16], this includes the power pills as "special dots". In our simplification of the game, power pills were simply removed, leaving 236 dots. At 10 points each, this means that clearing the maze of all dots will produce a score of 2360. The maze in the Ms. Pac-Man implementation by Lucas is slightly different, containing 220 dots plus 4 power pills [8].

B. Input Representation and Neural Network Model

In this paper we attempt to evolve agents to play Pac-Man based on a feedforward neural network (multi-layer perceptron) model acting as a controller. Each network has 4 output units representing the four possible directions (up, down, left and right) that Pac-Man can attempt to move in at any time-step of the game. Each output unit has a (logistic) sigmoidal activation function and the movement direction at each time-step is chosen according to the network output with the maximum value.

The majority of the input to the network is based on a window of the current on-screen information in the game. Consider a grid over the maze, with each cell at the level of detail of a position of a dot in the maze. In the implementation of the game used here, this leads to a 31×28 grid (including outer walls of the maze). The networks used take input from a window centered on the current location of Pac-Man in the maze (windows of sizes 5×5, 7×7 and 9×9 have been implemented). A wrap-around effect is implemented such that if the window ranges beyond the limits of the maze,

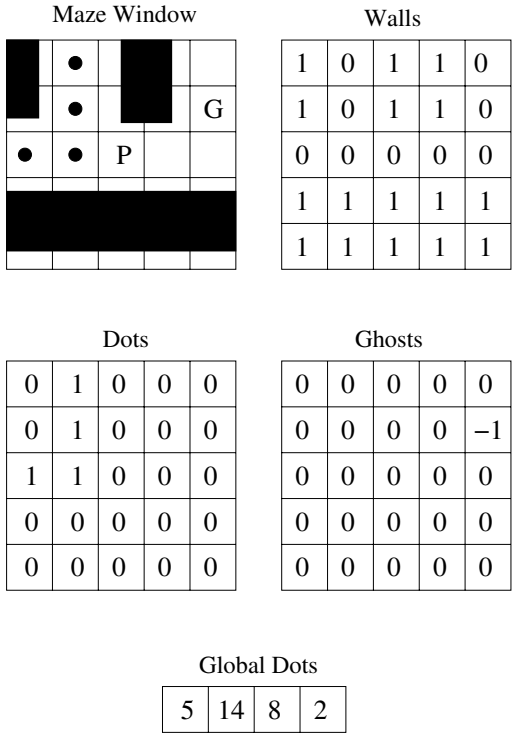

Fig. 2. Breakdown of the game representation used as input to the multi-layer perceptron networks. Shown (top-left) is an example of a maze view with a 5 × 5 window size, together with the corresponding inputs that would be produced from this maze view representing the walls, dots and ghosts within this window. Four additional inputs are included to provide global dot information beyond the current window.

view information from the opposite side(s) of the maze are represented. This negates the need to encode edge effects, as well as allowing for an effective representation of the tunnel in the center of the maze (which literally wraps around).

We encode three types of information from the maze input window (Figure 2). Walls and dots are each represented using a window-sized binary matrix of inputs. Ghosts are represented in a third matrix with a value of -1 while the absence of a ghost is indicated using a value of 0. When power pills are in play, a blue (edible) ghost can also be represented using a value of +1.

A potential side-effect of using a limited window size is that dots in the maze that do not appear in the current input window are effectively invisible to the agent. Four additional inputs are therefore used in the game representation. These represent the total amount of dots remaining in each of the four primary directions in the maze. The total number of inputs to each network is therefore $3w^2 + 4$, where w is the window height/width.

Multi-layer perceptrons with a single hidden layer were used, with logistic sigmoidal activation functions. The number of hidden units (and therefore the topology of the networks) was chosen prior to learning for each experiment and was therefore fixed.

C. Evolving the Neural Networks

The connection weights in the neural networks were evolved using a $(\mu + \lambda)$-Evolution Strategy. Mutation (without self-adaptation of parameters) was applied to each weight in a network with a given probability p_m, using a Gaussian mutation distribution with zero mean and standard deviation v_m. No recombination operator was used. This is a very simple EA, however it serves as a first attempt for conducting the experiments. The weights for the initial population of networks in each experiment were generated uniformly in the range $[0, 0.1]$.

The fitness function was also very simple (as also used in [8]) - the average number of points scored in a game (note that a game consists of three lives of Pac-Man) over the number of games played per agent, per generation

$$f = \frac{1}{N_{\text{games}}} \sum_{i=1}^{N_{\text{games}}} \text{score}$$

V. Results

A. One Deterministic Ghost

The first experimental scenario was the simplest possible. The game used a single ghost behaving deterministically (moving towards Pac-Man by the shortest path). In this case, the entire game dynamics are deterministic, hence N_{games} can be set to 1. Other parameters were as follows:

- Input window size: 5×5 (leading to networks with 79 inputs).
- Number of hidden units = 8.
- p_m = 0.1, v_m = 0.1.
- Population size: $\mu = 30, \lambda = 15$.

Three trials of this experiment were conducted. Figures 3-5 show the median (points) and best (line) fitness of each population over generations. The results indicate firstly that the methodology is able to evolve neural networks that improve their game-play over time. There is considerable variability between the three trials, with the highest fitness value obtained being 1290. This corresponds to clearing more than half of the first maze of dots.

Observing game-play reveals that the networks essentially learn patterns of clearing dots in the maze. In the early stages of evolution, low-scoring agents are those that quickly get stuck attempting to move in a direction where there is a wall. For example, from the starting position, Pac-Man can only move left or right - if the decision of the network output is to go up or down the agent will be stuck[1]. The results indicate that better performing agents become responsive to the *Walls* input window, since they move around the maze much more effectively. In higher scoring games at the end of the experiments, different paths were typically observed for different lives during a single game. This provides evidence that the behavior of agents was also responsive to changes in the *Dots* window inputs (since the disappearance of dots is the difference between the input patterns seen between lives, during a game. Responsiveness to the *Ghosts* input was much less evident from observing game-play of the fittest individuals. In certain situations, Pac-Man clearly did exhibit avoidance of the ghost. The clearest example was when Pac-Man was stuck at a location in the maze for several seconds, but when the ghost became very close (within the 5×5 input window size) the Pac-Man agent would begin moving again (typically away from the ghost!). However, it was also clear that the agents had not learnt a dominant "ghost-avoidance" strategy for play, which would effectively solve this simple version of the game.

[1]At least until something in the input changes.

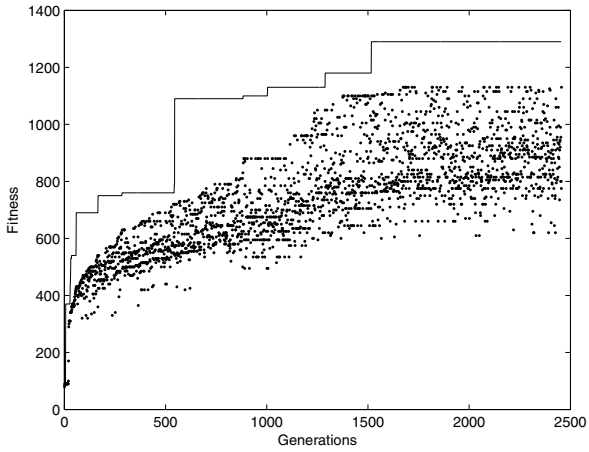

Fig. 3. Evolution of the agent population for the "one deterministic ghost" experiment (trial 1). The line shows the maximum fitness attained during each generation, while the points show the median fitness of each population.

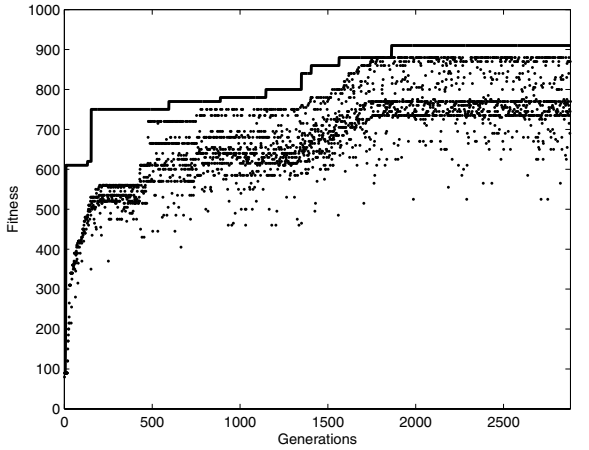

Fig. 4. Evolution of the agent population for the "one deterministic ghost" experiment (trial 2). The line shows the maximum fitness attained during each generation, while the points show the median fitness of each population.

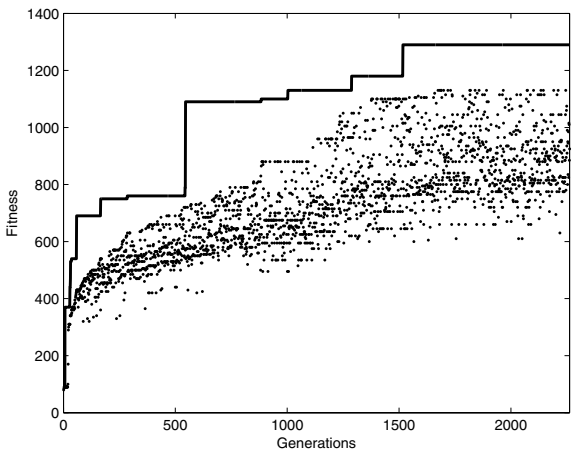

Fig. 5. Evolution of the agent population for the "one deterministic ghost" experiment (trial 3). The line shows the maximum fitness attained during each generation, while the points show the median fitness of each population.

B. More Complex Games and Varying Experimental Parameters

We also explored a number of different experimental configurations to see the effect (if any) on learning and performance of the agents. Here we attempt to summarize the interesting features of these experiments.

1) Non-deterministic Ghosts, Number of Hidden Units, Population Size, N_{games}: Several additional experiments were conducted using the 5×5 window size ($w = 5$). The deterministic ghost option was also removed, giving the game an element of randomness. The experimental configurations tested are summarized as follows:

- (e1): $w = 5$, $\mu = 10, \lambda = 5$, 3 hidden units, non-deterministic ghost, $N_{\text{games}} = 5$.
- (e2): $w = 5$, $\mu = 10, \lambda = 5$, 8 hidden units, non-deterministic ghost, $N_{\text{games}} = 5$.
- (e3): $w = 5$, $\mu = 50, \lambda = 25$, 3 hidden units, non-deterministic ghost, $N_{\text{games}} = 3$.
- (e4): $w = 5$, $\mu = 30, \lambda = 1$, 2 hidden units, 4 ghosts, $N_{\text{games}} = 2$.

Results for these experiments are shown in Table I. Clearly, none of these experiments produced impressive results. Although some degree of improvement always occurred, performance was significantly lower than for the "one deterministic ghost" experiments above. This is perhaps not surprising, since there is often a trade-off associated with these changes. For example, adding hidden units to the networks increases the potential representative power of the models, but also creates a higher-dimensional optimization problem for the EA, which may be more difficult to solve or may require significantly more generations. Increasing N_{games} should improve the quality of fitness measurements (when the game is stochastic), but increases the computation time required per generation.

TABLE I

SUMMARY OF RESULTS FOR EXPLORATORY EXPERIMENTS (BEST FITNESS VALUE ATTAINED, TOTAL NUMBER OF GENERATIONS).

Experiment	Best	Median	Generations
e1	770	630	2390
e2	510	400	2035
e3	710	460	4256
e4	700	640	513
e5	590	420	356
e6	690	550	1057
e7 (i)	590	420	366
e7 (ii)	590	440	732

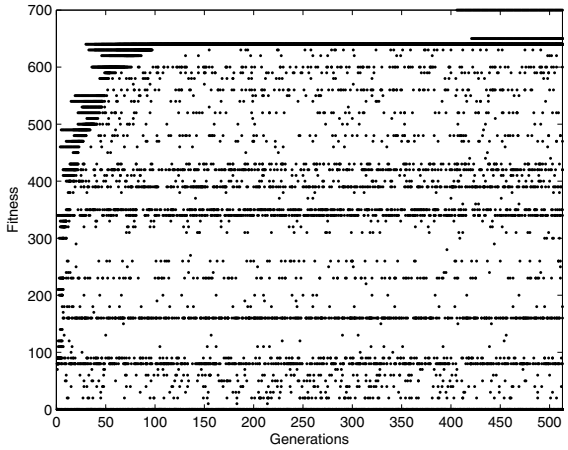

Fig. 6. Evolution of the agent population for experiment (e1). Fitness of each individual in the population is shown.

To illustrate the progress of learning, Figures 6 and 7 show the evolution of the population for experiments (e1) and (e4) respectively. Figure 6 shows considerable fitness diversity but highly variable learning progress. A relatively small population and the non-deterministic ghost behavior (despite 5 repeated games per fitness evaluation) are likely contributors to this variability. Note however that only 513 generations were completed for this experiment. Experiment (e4) used 4 ghosts in the maze with Pac-Man. The results show rapid initial learning but progress then appears to stall.

2) Larger Input Windows: Further experiments were conducted with larger input window sizes, more specifically:

- (e5): $w = 7$, 4 hidden units, non-deterministic ghost, $N_{\text{games}} = 5$.
- (e6): $w = 7$, 8 hidden units, non-deterministic ghost, $N_{\text{games}} = 5$.
- (e7): $w = 9$, 8 hidden units, non-deterministic ghost, $N_{\text{games}} = 3$ (2 trials).

Results are also summarized in Table I. A degree of learning was also observed in these experiments, but only produced modest performance. The increase in input dimension creates a population of larger networks, so it seems possible that many more generations might be required to allow the EA reasonable time to search the model space.

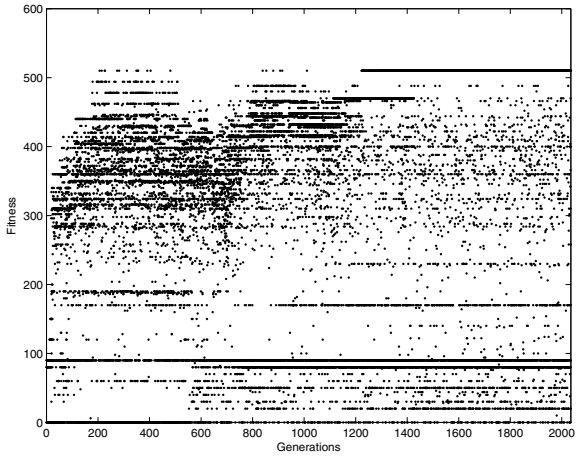

Fig. 7. Evolution of the agent population for experiment (e4). Fitness of each individual in the population is shown.

VI. Discussion

The aim of this paper was to examine the feasibility of evolving a Pac-Man playing agent based on limited input information. The results demonstrate that it is possible to use neuroevolution to produce Pac-Man playing agents (albeit in a simplified game) with basic playing ability using a minimal amount of raw on-screen information. The performance obtained to date is lower than previous approaches [6], [8]; no agent was able to clear a maze of dots in our experiments. Nevertheless, it is encouraging and perhaps surprising that it is possible to learn anything at all given the limited representation of the game, feedback about performance and incorporation of prior knowledge. More complex experimental configurations were considered, but the results are inconclusive with respect to the influence of a number of system parameters on performance.

The approach taken in this paper is simplistic in several respects and there are a number of clear directions that could be taken to try and improve the work. The computation time involved with our version of the game is significant, despite some attempt to optimize the running speed. Further experiments are expected to require thousands of generations to obtain clear results, but this currently takes large amounts of time. The fixed-topology neuroevolution implemented above is a "bare-bones" approach and significant performance improvements might be gained by considering more powerful approaches [13].

The representation of the game in this work is minimalist. In fact it could be argued that *too* little information is provided (e.g. requiring the networks to learn to move instead of running into walls). On the other hand, it may be possible to simplify the representation even further. For example, we have not studied the influence of the "Global Dots" inputs on performance: it is possible that similar performance can be achieved without this information.

There is a trade-off between incorporating little knowledge of the game into the representation/model (making it impossible to learn), versus providing too much prior information (making the learning task trivial and the resulting game-play uninteresting). Finally, subtle differences in game implementations may have unexpected effects on results, which may hamper the progress of research it terms of the comparability and repeatability of approaches. Despite these concerns, Pac-Man provides a useful testbed for Computational Intelligence in games and there is significant scope for further research in this domain.

References

[1] K. Uston, *Mastering Pac-Man*. Macdonald and Co, 1981.
[2] J. Koza, *Genetic Programming: On the Programming of Computer by Means of Natural Selection*. MIT Press, 1992.
[3] J. P. Rosca, "Generality versus size in genetic programming," in *Genetic Programming (GP96) Conference*, J. K. et. al., Ed. MIT Press, 1996, pp. 381–387.
[4] A. Kalyanpur and M. Simon, "Pacman using genetic algorithms and neural networks," Retrieved from http://www.ece.umd.edu/∼adityak/Pacman.pdf (19/06/03), 2001.
[5] J. S. De Bonet and C. P. Stauffer, "Learning to play pac-man using incremental reinforcement learning," Retrieved from http://www.debonet.com/Research/Learning/PacMan/ (20/10/06).
[6] M. Gallagher and A. Ryan, "Learning to play Pac-Man: An evolutionary, rule-based approach," in *Congress on Evolutionary Computation (CEC)*, R. S. et. al., Ed., 2003, pp. 2462–2469.
[7] S. Baluja, "Population-Based Incremental Learning: A method for integrating genetic search based function optimization and competitive learning," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-94-163, 1994.
[8] S. M. Lucas, "Evolving a neural network location evaluator to play Ms. Pac-Man," in *IEEE Symposium on Computational Intelligence and Games*, 2005, pp. 203–210.
[9] S. M. Lucas and G. Kendall, "Evolutionary computation and games," *IEEE Computational Intelligence Magazine*, vol. 1, no. 1, pp. 10–18, 2006.
[10] K. Chellipilla and D. Fogel, "Evolving an expert checkers playing program without using human experience," *IEEE Transactions on Evolutionary Computation*, vol. 5, pp. 422–428, 2001.
[11] D. Fogel, *Blondie24: Plyaing at the Edge of AI*. Morgan Kaufmann, 2001.
[12] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002.
[13] K. O. Stanley, B. D. Bryant, and R. Miikkulainen, "Evolving neural network agents in the NERO video game," in *IEEE Symposium on Computational Intelligence and Games*, 2005.
[14] ——, "Real-time neuroevolution in the NERO video game," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 6, pp. 653–668, 2005.
[15] B. Chow, "Pac-man (java implementation)," Retrieved from http://www.bennychow.com (26/10/06), 2000.
[16] Various, "Pac-man (wikipedia article)," Retrieved from http://en.wikipedia.org/wiki/Pac-man (26/10/06).