

Learning to Play Pac-Man: An Evolutionary, Rule-based Approach

Marcus Gallagher
marcusg@itee.uq.edu.au

Amanda Ryan
s354299@student.uq.edu.au

School of Information Technology and Electrical Engineering
University of Queensland 4072
Australia

Abstract- Pac-Man is a well-known, real-time computer game that provides an interesting platform for research. This paper describes an initial approach to developing an artificial agent that replaces the human to play a simplified version of Pac-Man. The agent is specified as a simple finite state machine and ruleset, with parameters that control the probability of movement by the agent given the constraints of the maze at some instant of time. In contrast to previous approaches, the agent represents a dynamic strategy for playing Pac-Man, rather than a pre-programmed maze-solving method. The agent adaptively “learns” through the application of population-based incremental learning (PBIL) to adjust the agents’ parameters. Experimental results are presented that give insight into some of the complexities of the game, as well as highlighting the limitations and difficulties of the representation of the agent.

1 Introduction

Pac-Man is a well-known, real-time arcade computer game originally developed by Toru Iwatani for the Namco Company in 1981. Different versions of the game have been developed subsequently for a large number of home computer, game-console, and hand-held systems.

The typical version of Pac-Man is a one-player game where the human player maneuvers the Pac-Man character around a maze, attempting to avoid four “ghost” characters while eating dots initially distributed throughout the maze. 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 centre 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 of time. The game ends when Pac-Man has lost all (usually 3) of his lives. Figure 1 shows a screen-shot of the starting position of the first maze in the game. Pac-Man is a real-time computer game that resembles a simplified version of many modern first-person environment computer games. That is, the game is centered around navigating the player character around a semi-structured world, accumulating points, avoiding and (when appropriate) attacking

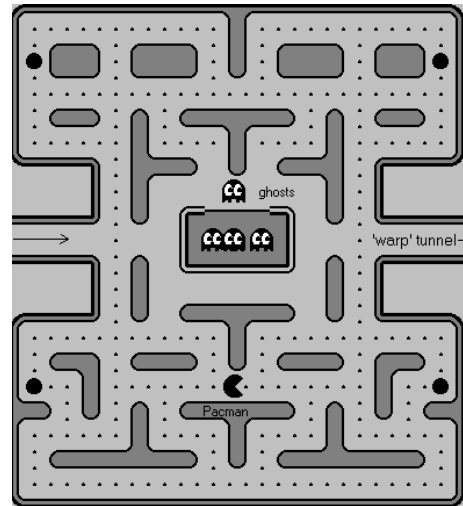


Figure 1: The starting position of the Pac-Man game, showing the maze structure, Pac-Man (lower-center), power pills (large dots), dots (small dots) and ghosts (center).

non-player game characters¹. This kind of game requires dynamic control of the game agent by the human player and involves task prioritization, planning, and risk assessment. While it is relatively easy for a human to learn a basic strategy for Pac-Man, the game has complex aspects that allow the possibility of developing more intelligent strategies (in conjunction with other skills such as hand-eye coordination). It is also a challenge for a person to describe precisely their Pac-Man-playing strategy, or to represent such a strategy formally (e.g., as a set of rules).

This paper describes an initial approach to developing an artificial agent that replaces the human playing Pac-Man. For this work, the game has been significantly simplified to having only a single ghost, dots and the Pac-Man agent present in the mazes. The agent is specified as a simple finite state machine and ruleset, with parameters that specify the state transition and the probabilities of movement according to each rule. Section 2 provides an overview of previous work relevant to Pac-Man and AI game playing,

¹ Alternatively, other game characters might be controlled by other human players in a multi-player setting.

while Section 3 describes the evolutionary, rule-based approach taken in this paper. Some experimental details are discussed in Section 4. Results are presented in Section 5, and Section 6 provides a summary and some discussion of possible future work.

2 Artificial Intelligence Approaches to Pac-Man

A relatively small amount of previous research has been done toward the application of artificial intelligence to Pac-Man or similar games. Koza [7] and Rosca [9] 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. Genetic Programming has also been applied to the computer game Tron [3], where coevolution was used to produce agents that learn strategies by playing human opponents over the internet.

Gugler [5] describes “Pac-Tape”, a project which attempted to produce a self-playing Pac-Man based on the original Pac-Man arcade game emulated on a desktop PC. The approach appears to be based on brute force search, but is not described or tested. Lawrence [8] builds on the Pac-Tape work, but applies a genetic algorithm to evolve a string of directions (north, south, east, west) to traverse a maze. The aim was to evolve interesting patterns for solving particular mazes. Unfortunately, success was limited, perhaps due to unfavourable interactions between the genetic operators (esp. crossover) and the representation used.

Kalyanpur and Simon [6] 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 [2] 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.

The aim of our research is to investigate techniques to develop an adaptive agent that learns inductively to play Pac-Man. Our approach is aimed at producing agents that can learn based only on the information that is available to a human playing the game (or quantities that a human could conceivably estimate in real-time). In particular, the agent should not be able to exploit internal knowledge about the game (e.g. the programmed behaviour of the ghosts) - the (software) agent should be considered external from the game software. Furthermore, this agent should learn gen-

eralizable strategies for good game play, based not on the exact structure of a given maze but rather on more general principles of good strategies to play Pac-Man.

3 Designing an Artificially Intelligent Agent that learns to play Pac-Man Inductively

Pac-Man is a simple game to describe and play. Human players have little difficulty in learning the basics of game play, which are to obtain as many points as possible (by clearing mazes of dots and eating power pills and ghosts), whilst avoiding the ghosts. However there is not obviously one correct way of specifying such a strategy precisely, perhaps because it is largely based on perception of the visual presentation of the game and its dynamics in time.

Moving beyond this basic approach makes clear the full complexity of the game domain. A large point incentive is provided for eating power pills, followed by eating ghosts in the few seconds that the power pill remains effective, but the ghosts generally try to retreat from Pac-Man in this time to avoid being eaten. Thus an effective strategy might involve attracting ghosts to the area of the maze near a power pill before eating it. The ghosts movements are dependent on each other and typically contain an element of randomness, making it difficult to predict their behaviour and plan around it.

Our approach here is to initially remove much of the complexity from the game to see if it is possible to produce an effective agent in a simplified Pac-Man environment. In the experiments discussed below, only the dots and a single ghost are present in a maze with Pac-Man. While this clearly removes several interesting aspects of game-play, we believe that what remains (learning to avoid the ghost and perhaps to include random exploration) is still a fundamental aspect of the full Pac-Man game.

We were motivated to begin by using a simple, transparent representation for the agent. A representation that was potentially capable of capturing the general aspects of a basic human strategy provides a good platform on which to test the feasibility of using an evolutionary approach to learning, as well as serving as a benchmark for comparing future work. It was also interesting to test if the evolutionary algorithm would generate an agent with an obvious strategy. These considerations lead to the agent being represented as a two-state finite state machine, with a set of probabilistic rules for each state that control the movement of the agent. At any given time, the state of the agent is determined by the distance between Pac-Man and the ghost. If this distance is greater than some given value, p_1 , the agent is in the *Explore* state, while if the distance is less than p_1 , the agent switches to the *Retreat* state.

A human player is able to see the entire maze during game play, but dynamically is usually focusing their atten-

tion primarily on the immediate area of the maze where Pac-Man is currently located. Planning a long sequence of moves in advance would not only be very difficult for a human, but also ineffective, since the dynamics of the game would be likely to make such a strategy poor and irrelevant. Our initial approach for an artificial Pac-Man agent is therefore to view play as a small number of possible turn types, together with basic global information such as the position and movement of the ghost. Figure 2 categorizes the entire maze in terms of turn types, including straight corridors. Note that the representation used below considers turn types from the current orientation of Pac-Man, so that the “true” orientation of a turn type does not need to be directly considered (e.g., vertical versus horizontal corridors).

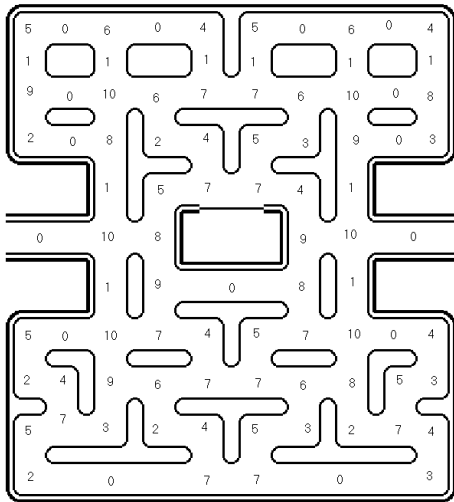


Figure 2: The first Pac-Man maze in terms of turn types. Some types are more common than others.

At each time step (tick) of the game, the agent is located in an instance of a certain turn-type, and needs to produce an output that becomes the current direction of movement for Pac-Man. Depending on the turn-type of the current position (corridor, L-turn, T-junction, intersection), there are a number of different feasible directions to move, including maintaining the current direction (but excluding running directly into walls). In the Explore state, a probabilistic decision is made to choose one of the feasible directions depending on the turn type. As a result, Pac-Man is able to explore the maze in a pseudo-random fashion. Exploration does not use any other game information to determine the agent’s output. In the Retreat state, the agent additionally considers the location of the ghost to determine the output. This results in the need to consider a number of possible turn-type and ghost position combinations. We classified the ghost’s position as being either forward, back, left, right, forward-left, forward-right, back-left, or back-right relative

```

Agent() {
  while (game is in play)
    currentdistance = Distance(pacman,ghost)
    if currentdistance > p1 then
      Explore
    else
      Retreat
  end
}
Explore() {
  switch(turntype) {
    case "corridor"
      newdir = Random(P,prevdir,turntype)
    case "L-turn"
      newdir = Random(P,prevdir,turntype)
    case "T-turn"
      newdir = Random(P,prevdir,turntype,orientation)
    case "Intersection"
      newdir = Random(P,prevdir,turntype)
  }
}
Retreat() {
  switch(turntype) {
    case "corridor"
      newdir = Random(P,prevdir,turntype,ghostpos)
    case "L-turn"
      newdir = Random(P,prevdir,turntype,orientation,ghostpos)
    case "T-turn"
      newdir = Random(P,prevdir,turntype,orientation,ghostpos)
    case "Intersection"
      newdir = Random(P,prevdir,turntype,ghostpos)
  }
}

```

Figure 3: Pseudo-code for the Pac-Man agent strategy.

to Pac-Man. Note that the first four classifications are determined using only one coordinate dimension. For example, if Pac-Man is currently facing north (up) on the screen, the ghost would be classified as being “back” if its current y-coordinate is less than Pac-Man’s current y-coordinate. Adding the final four classifications are one way of refining this information. For the same example (Pac-Man facing north), the ghost’s location will be classified as follows:

- **back** if $(ghost_y < pacman_y)$ and $(ghost_x = pacman_x)$
- **back-left/left-back** if $(ghost_y < pacman_y)$ and $(ghost_x < pacman_x)$
- **back-right/right-back** if $(ghost_y < pacman_y)$ and $(ghost_x > pacman_x)$
- **forward** if $(ghost_y > pacman_y)$ and $(ghost_x = pacman_x)$
- **forward-left/left-forward** if $(ghost_y > pacman_y)$ and $(ghost_x < pacman_x)$
- **forward-right/right-forward** if $(ghost_y > pacman_y)$ and $(ghost_x > pacman_x)$
- **left** if $(ghost_y = pacman_y)$ and $(ghost_x < pacman_x)$
- **right** if $(ghost_y = pacman_y)$ and $(ghost_x > pacman_x)$

Note also that, for example, back-left is equivalent to left-back as indicated in the above. Unfortunately, having eight classes for the position of the ghost for each turn-type means that the ruleset (and the number of parameters) also gets larger. Hence in our implementation, eight classes were used to classify the ghost’s position for intersections, with only the first four classes used for other turn-types. Pseudocode for the agent’s strategy is shown in Figure 3.

Implementation of the agent based on the specifications above leads to a total of 85 adjustable parameters, collected into a parameter vector $P = (p_1, \dots, p_{85})$, that controls the agent’s behaviour. A sub-component of P is used to decide Pac-Man’s current (new) direction of movement, depending on the previous direction (prevdir), turn type, orientation, and (in retreat mode) the ghost’s location. Parameter p_1 is the distance value at which the agent shifts from Explore to Retreat (and vice-versa), calculated as the direct (i.e., ignoring maze walls) Manhattan distance between Pac-Man and the ghost. All other parameters of P are probability values, $0 \leq p_i \leq 1, i = 2, \dots, 85$. Table 1 gives a complete specification of these values.

Parameter	Description
p_1	Distance to ghost
Explore:	
p_{2-3}	Corridor: forward, backward
p_{4-5}	L-turn: forward, backward
p_{6-8}	T-turn (a) approach centre
p_{9-11}	T-turn (b) approach left
p_{12-14}	T-turn (c) approach right
p_{15-18}	Intersection
Retreat:	
p_{19-20}	Corridor: ghost forward
p_{21-22}	Corridor: ghost behind
p_{23-24}	L-turn: ghost forward
p_{25-26}	L-turn: ghost behind
p_{27-29}	T-turn (a): ghost behind
p_{30-32}	T-turn (b): ghost behind
p_{33-35}	T-turn (c): ghost behind
p_{36-38}	T-turn (a): ghost on left
p_{39-41}	T-turn (b): ghost on left
p_{42-44}	T-turn (a): ghost on right
p_{45-47}	T-turn (b): ghost on right
p_{48-50}	T-turn (b): ghost forward
p_{51-53}	T-turn (c): ghost forward
p_{54-57}	Intersection : ghost forward
p_{58-61}	Intersection : ghost behind
p_{62-65}	Intersection : ghost left
p_{66-69}	Intersection : ghost right
p_{70-73}	Intersection : ghost forward/left
p_{74-77}	Intersection : ghost forward/right
p_{78-81}	Intersection : ghost behind/left
p_{82-85}	Intersection : ghost behind/right

Table 1: Description of parameter values used in the Pac-Man agent and their usage.

4 Simulation Methodology

4.1 Fitness Function

For our simplified Pac-Man game, a fitness function was developed based primarily on the score obtained by Pac-Man. The only way to score points in this simplified game is to eat dots from the maze, and since there is a fixed initial number of dots in a maze, the maximum possible score for each maze is known *a priori*. It was considered that the time Pac-Man manages to survive should also be a factor in the fitness function. Note however that there also exists a possibility that Pac-Man may successfully avoid the ghost indefinitely, but fail to clear the maze of dots. Because of this a term was added to reward the time survived by Pac-Man, but imposing a pre-chosen limit on this factor. The fitness function used is:

$$f = \sum_{\text{level}} \frac{\text{score}_{\text{level}}}{\text{maxscore}_{\text{level}}} + \min \left\{ \frac{\text{time}_{\text{level}}}{\text{maxtime}_{\text{level}}}, 1 \right\} \quad (1)$$

The score and time factors in the fitness function are normalized by their maximum (respectively, known and chosen) values for each level. Hence, the maximum fitness for each level is 2.

The movement of the ghost in the game has a small amount of randomness, and the agent developed above clearly produces stochastic movement for Pac-Man. As a consequence, the fitness value of each game given a fixed P will also be stochastic. In such a situation it is typical to conduct repeated evaluations of the fitness function with a given parameter set, and produce an average fitness value to be used by the evolutionary algorithm. In the simulations below, 10 games were played per average fitness evaluation. Recall that Pac-Man has three lives in a game, meaning that Pac-Man “runs” in the maze 30 times for each fitness evaluation used in the EA.

4.2 Algorithm

The algorithm used to learn the parameters of an agent is an implementation of population-based incremental learning (PBIL) for continuous search spaces [1, 10, 11]. PBIL replaces the conventional genetic operators of mutation and recombination with a probability vector, which is used to generate each population and is updated via a learning rule based on the single best individual in the current population. Each component of the probability vector represents the mean of a Gaussian distribution. In our experiments the population size was set to 25 (computational time prevented a larger value). For the standard deviations of the Gaussians and the PBIL learning rate parameter, α , several different values were tested. Note that for $\alpha = 1$, this algorithm is equivalent to a simple $(1, \lambda)$ evolution strategy with a constant standard deviation value for all variables [4].

Vector	Mean	Std. Dev.	Min.	Max.
P_{h1}	0.215	0.080	0.104	0.511
P_{h2}	0.613	0.316	0.187	1.699
P_{h3}	1.372	0.522	0.276	2.062

Table 2: Results for hand-coded parameter vectors tested, summarizing fitness values from 50 games with each parameter vector.

The probability vector was initialized with $p_1 = 15$ - this value p_1 chosen such that the agent initially spent roughly equal amounts of time in the Explore and Retreat states. p_1 was allowed to evolve as a continuous value, although the Manhattan distance between Pac-Man and the ghost is integer-valued. The remaining parameters represented probabilities for subsets of variables, so they were initialized with uniform probability for each feasible output direction of the agent (e.g. $p_{2-3} = 0.5$, $p_{15-18} = 0.25$). These parameters were re-normalized after being updated at each generation by the PBIL learning rule. Note that PBIL evolves a probabilistic model of the search space which converges towards a locally or globally optimal value. In this paper we interpret the probability vector learnt by the algorithm as the realization of our evolved “agent”, and a population-based search is used to perform this task.

5 Results

5.1 Hand-coded Agent Parameters

A feature of the agent implementation described above is that it is transparent: each parameter has a clear function in the control of the output of the agent. It is therefore possible to experiment with an agent of hand-coded parameters. The rule sets allow for basic explore and retreat behaviour only, so the hand-coded parameters were chosen to try and produce an agent with the ability to explore the maze widely in the “Explore” state, and to retreat from the ghost to avoid being eaten in the “Retreat” state. Results for the different hand-coded parameter vectors tested are shown in Table 2.

Consider firstly an agent, P_{h1} , with equal probabilities assigned to each parameter in each relevant subset of values. The behaviour of this agent was observed to be highly erratic, because equal probabilities mean that Pac-Man is just as likely to reverse his direction at any point in time as he is to choose any other feasible direction. As a result, the movement of Pac-Man is dominated by oscillations (e.g., forwards-backwards in a corridor) around the current point. This is reflected in the low fitness values observed for agent P_{h1} in Table 2.

For the next parameter vector tested (P_{h2}), in the Explore state a lower probability was assigned to reversing the current direction, with probabilities for all other feasible di-

rections assigned uniformly (e.g. for a corridor to move forwards and backwards respectively, $p_2 = 0.8, p_3 = 0.2$; for an intersection to move forwards, backwards, left and right respectively $p_{15,17,18} = 0.3, p_{16} = 0.1$). For the Retreat state, zero probability was assigned to the direction that led to the ghost, with all other feasible options given uniform probability. This agent had reduced oscillations in its movement and was observed to have some ability to retreat from the ghost over “chase” sequences of turns (see Table 2 for results). However, the agent only occasionally comes close to clearing a maze (surviving long enough to do so) and still contains a harmful degree of oscillation in its output.

Finally, a parameter vector (P_{h3}) was tested that refined P_{h2} by having very low probability values for reversing the current direction. For example, for a corridor to move forwards and backwards respectively, $p_2 = 0.99, p_3 = 0.01$; for an intersection to move forwards, backwards, left and right respectively $p_{15,17,18} = 0.33, p_{16} = 0.01$. This agent produced improved behaviour (Table 2), typically coming close to clearing the maze and occasionally doing so (6 out of the 50 trials produced fitness values above 2.0). More importantly (since the Explore state for the agent does not have any knowledge of the dots in the maze), the agent is able to evade the ghost for long periods of time, due to the Retreat state. Nevertheless, the limitations of the agent became clear from observing the performance of this agent. The ruleset decides the move direction based on the current location only. In retreat mode, the position of the ghost is considered but only crudely measured. As an example, consider a situation where Pac-Man is located at one of the extreme corners of the maze (ref. Figure 1 or Figure 2). For say, the top-right corner, the ghost will usually be moving towards Pac-Man from a below-left position. Because of the limited way that the agent considers the position of the ghost, moving left is undesirable because the ghost is to the left, but moving downward is also undesirable because the ghost is also downward. Oscillation results, until the ghost becomes very close to Pac-Man. Other situations allow the agent to be captured by the ghost because equiprobable alternatives for retreating can lead to oscillatory behaviour.

5.2 PBIL-evolved Agent

Figure 4 shows the result of the PBIL algorithm evolving a Pac-Man agent parameter vector, in terms of the minimum, mean, and maximum fitness values in the population distribution. Although the game code was modified to make play as fast as possible, 250 games are played in each generation of the algorithm (population of 25, 10 games per fitness evaluation), meaning several days of simulation time. For this experiment, a standard deviation of 0.01 was used for all parameters except p_1 , which used a value of 1.0, and the learning rate was $\alpha = 0.05$. The mean of the population

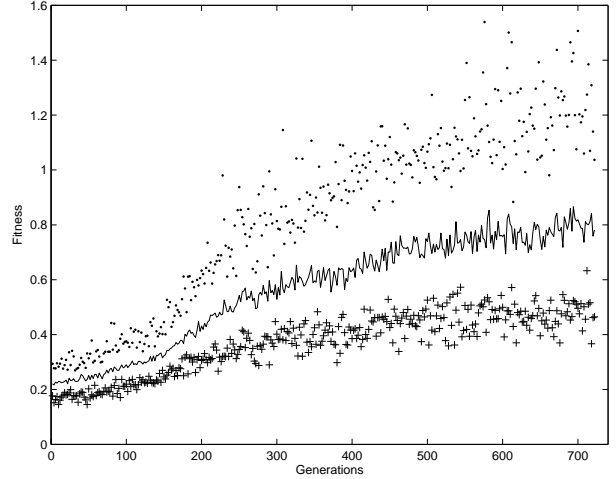


Figure 4: Evolution of the minimum (crosses), mean (line) and maximum (dots) fitness values in the population for PBIL with $\alpha = 0.05$.

shows a smooth learning curve compared to other standard deviation/learning rate values that we experimented with. After 722 generations the population mean is around 0.8, which is better than our hand-coded parameter sets P_1 and P_2 but below the performance of P_3 . We were interested in the influence of the PBIL learning rate on the results of the algorithm. Figure 5 shows the result of a different experiment, with $\alpha = 1.0$ (note that less generations have been performed). This is equivalent to a $(1, 25)$ -evolution strategy. It is evident that this algorithm is able to initially provide more rapid improvement in fitness, but after approximately 100 generations progress is much more noisy than that of Figure 4. In preliminary experiments we observed that a larger standard deviation value had a similar effect. The results suggest that a kind of annealing scheme for either of these parameters may allow the algorithm to converge more reliably. Overall the performance of the algorithms with different learning rates is similar in terms of the kind of fitness values obtained during learning. The probability vectors produced by the the two algorithms in the experiments above were examined manually. Many of the parameters appeared to be converging towards values similar to the hand-coded strategy P_{h3} above. It is clear from the results shown in Figures 4 and 5 that the probability vector is still subject to significant perturbation, making it difficult to interpret without extending the experiment over more generations.

6 Discussion

This paper has developed an approach to constructing an artificial agent that learns to play a simplified version of

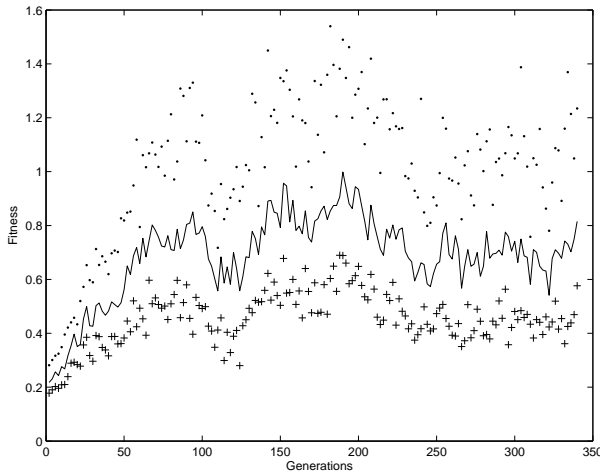


Figure 5: Evolution of the minimum (crosses), mean (line) and maximum (dots) fitness values in the population for PBIL with learning rate = 1.0 (i.e. (1,25)-ES).

Pac-Man. The agent is specified as a simple state machine and parameterized ruleset, with the PBIL algorithm used to learn suitable values for these parameters. Hand-coded parameters were also tested.

The results highlight the limitations of the representation used as well as some of the complexities of the game, even in the highly simplified form considered in these experiments. The ruleset could certainly be improved given the difficulties observed in our simulations. For example, considering 8 classifications for the ghost's position for each turn type (Section 3 should reduce the problems observed in Section 5.1. This would however necessarily result in a larger ruleset. The representation also has redundancies in the way that probability values are represented (e.g. replace p_3 by $(1 - p_2)$). Nevertheless, the representation seems to have serious limitations for scaling up the "intelligence" of the agent. This may result in a very high dimensional optimization problem which would require a large amount of computation time to allow enough generations for the algorithm to produce a good result. Given the complexities of the full game compared to the version considered here, it seems that extending on a ruleset-based representation may be difficult and impractical.

The methodology used in this paper is a first attempt at this problem and there are several factors that might be improved. Firstly, it is clear that the different strategies (hand-coded and learned) used in this paper do not come close to capturing the variety of possible effective strategies used by humans in the full Pac-Man game. We hypothesize that the basic explore/retreat approach is likely to have a role in an effective strategy for the full Pac-Man game, but this has not been verified. Secondly, our decision to use an agent

that can consider only "localized" information of the game is a factor that deserves further consideration. Finally, the fitness function used is based on both score achieved and time taken. The impact of the time factor on our results is not clear and it may be possible to remove this from the fitness function with no adverse effects.

Nevertheless, we believe that the approach taken here could be useful as a benchmark in considering different representations and approaches to evolving a Pac-Man playing agent. It is expected that future work would need to be able to improve on the performance (fitness) of the agent. However it will also be interesting to compare the complexity (dimensionality, interpretability, computational requirements) of alternative approaches, to the rule-based approach developed above. Finally, we conjecture that general aspects of the approach taken here to developing an adaptive agent for Pac-Man may eventually lead to techniques that can be applied successfully to other real-time computer games.

Bibliography

- [1] S. Baluja. Population-Based Incremental Learning: A method for integrating genetic search based function optimization and competitive learning. Technical Report CMU-CS-94-163, School of Computer Science, Carnegie Mellon University, 1994.
- [2] J. S. De Bonet and C. P. Stauffer. Learning to play pacman using incremental reinforcement learning. Retrieved from <http://www.ai.mit.edu/people/stauffer/Projects/PacMan/> (19/06/03), 1999.
- [3] P. Funes, E. Sklar, H. Juillé, and J. Pollack. Animal-animat coevolution: using the animal population as fitness function. In R. Pfeifer et. al., editor, *From Animals to Animats 5: Proceedings of the Fifth International Conference on Simulation of Adaptive Behaviour*, pages 525–533. MIT Press, 1998.
- [4] M. Gallagher. *Multi-layer perceptron error surfaces: visualization, structure and modelling*. PhD thesis, Dept. Computer Science and Electrical Engineering, University of Queensland, 2000.
- [5] S. Gugler. Pac-tape. Retrieved from <http://www.geocities.com/SiliconValley/Bay/4458/> (21/05/01), 1997.
- [6] 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.

- [7] J. Koza. *Genetic Programming: On the Programming of Computer by Means of Natural Selection*. MIT Press, 1992.
- [8] S. Lawrence. Pac-man autoplayer. Retrieved from <http://www.cis.rit.edu/~jerry/Software/autopac/> (21/05/01), 1999.
- [9] J. P. Rosca. Generality versus size in genetic programming. In J. Koza et. al., editor, *Genetic Programming (GP96) Conference*, pages 381–387. MIT Press, 1996.
- [10] S. Rudlof and M. Köppen. Stochastic hill climbing with learning by vectors of normal distributions. In *1st Online Workshop on Soft Computing*, Retrieved from <http://www.bioele.nuee.nagoya-u.ac.jp/wsc1/> (8/12/99), 1996.
- [11] M. Sebag and A. Ducoulombier. Extending population-based incremental learning to continuous search spaces. In A. Eiben et. al., editor, *Parallel Problem Solving from Nature - PPSN V*, volume 1498 of *Lecture Notes in Computer Science*, pages 418–427, Berlin, New York, 1998. Springer.