Evolving a Ms. PacMan Controller using Grammatical Evolution

Edgar Galván-López, John Mark Swafford,
Michael O'Neill and Anthony Brabazon

Natural Computing Research & Applications Group,
University College Dublin, Ireland
edgar.galvan, john-mark.swafford, m.oneill, anthony.brabazon @ ucd.ie

Abstract. In this paper we propose an evolutionary approach capable of successfully combining rules to play the popular video game, Ms. Pac-Man. In particular we focus our attention on the benefits of using Grammatical Evolution to combine rules in the form of "if <*condition*> then perform <*action*>". We defined a set of high-level functions that we think are necessary to successufully maneuver Ms. Pac-Man through a maze while trying to get the highest possible score. For comparison purposes, we used four Ms. Pac-Man agents, including a hand-coded agent, and tested them against three different ghosts teams. Our approach shows that the evolved controller achieved the highest score among all the other tested controllers, regardless of the ghost team used.

1 Introduction

Ms. Pac-Man, released in early 1980s, became one the most popular video games of all time. This game, the sequel to Pac-Man, consists of guiding Ms. Pac-Man through a maze, eating pills, power pills, and fruit. This task would be simple enough if it was not for the presence of four ghosts that try to catch Ms. Pac-Man. Each ghost has their own, well-defined, behaviour. These behaviors are the largest difference between the Pac-Man and Ms. Pac-Man. In the original Pac-Man, the ghosts are deterministic and players who understand their behavior may always predict where the ghosts will move. In Ms. Pac-Man, the ghosts have non-deterministic elements in their behavior and are not as predictable.

The gameplay mechanics of Ms. Pac-Man are also very easy to understand. When Ms. Pac-Man eats a power pill, the ghosts change their status from inedible to edible (only if they are outside their "nest", located at the centre of the maze) and remain edible for a few seconds. In the edible state they are defensive, and if they are eaten, Ms. Pac-Man's score is increased considerably (the first eaten ghost gives 200 points, the second 400, the third 800, and 1,600 for the last). When all the pills are eaten, Ms. Pac-Man is taken to the next level. Levels get progressively harder by changing the maze, increasing the speed of the ghosts, and decreasing the time to eat edible ghosts. The original version of Ms. Pac-Man presents some very interesting features. For instance, Ms. Pac-Man moves slightly slower than Ghosts when she's eating pills, but she moves slightly faster

when crossing tunnels. The most challenging element is the fact that the ghosts' movements are non-deterministic. The goal of the ghosts is to catch Ms. Pac-Man, so they are designed to attack her. Over the last few years, researchers have tried to develop software agents able to successfully clear the levels and simultaneously get the highest score possible (the world record for a human player on the original game stands at 921,360 [5]). The highest score achieved by a computer, developed by Matsumoto [6], based on a screen-capture system that is supposed to be exactly the same as the arcade game, stands at 30,010 [5]. The other top three scores achieved are 15640, 9000 and 8740 points, respectively [6]. It is worth pointing out that all of these methods used a hand-coded approach

However, it is important to note that there has been work where researchers have used a variety of artificial intelligence approaches to create Ms. Pac-Man players. Some of these approaches state a goal of evolving the best Ms. Pac-Man player possible. Others aim to study different characteristics of an algorithm in the context of this non-deterministic game. Some previous approaches are listed here, but will not be compared against each other due to differences in the Ms. Pac-Man implementation and the goal of the approach.

One of the earliest, and most relevant, approaches comes from Koza [3]. He used genetic programming to combine pre-defined actions and conditional statements to evolve his own, simple Ms. Pac-Man game players. Koza's primary goal was to achieve the highest possible Ms. Pac-Man score using a fitness function that only accounts for the points earned per game. Work similar to [3] is reported by Szita and Lõrincz [10]. Their approach used a combination of reinforcement learning and the cross-entropy method to assist the Ms. Pac-Man agent in "learning" the appropriate decisions for different circumstances in the game. More evolution of Ms. Pac-Man players was carried out by Gallagher [2]. He used a population-based incremental learning approach to help one Ms. Pac-Man player "learn" how to improve its performance by modifying its different parameters. Another, more recent, approach by Lucas [4] uses an evolutionary strategy to train a neural network to play Ms. Pac-Man in hopes of creating the best possible player.

The goal of our work is to successfully evolve rules in the form of "if <*condition*> then perform <*action*>" to maneuver Ms. Pac-Man through the maze, and at the same time, achieve the highest score possible. For this purpose we are going to use Grammatical Evolution (GE) [8, 1].

This paper is structured as follows. In the following section we describe how GE works. In Sect. 3 we describe the high-level functions designed to evolve the Ms. Pac-Man agent. In Sect. 4 we describe the experimental setup and Sect. 5 presents the results achieved by our approach, followed by a discussion. Finally, Sect. 6 draws some conclusions.

2 Grammatical Evolution

In GE, rather than representing programs as parse trees, as in Genetic Programming (GP) [3], a variable length linear genome representation is used. This

genome is an integer array with elements called *codons*. A genotype to phenotype mapping process is employed on these integer arrays which uses a user-specified grammar in Backus-Naur Form (BNF) to output the actual phenotype. A grammar can be represented by the tuple $\{N, T, P, S\}$, where N is the set of non-terminals, T is the terminal set, P stands for a set of production rules and, S is the start symbol which is also an element of N. It is important to note that N may be mapped to other elements from N as well as elements from T. The following is an example based on the grammar used in this work (Note: the following is not the actual grammar, just a simplified version; see Fig. 2 for the actual grammar):

```
    Rule                      Productions                  Number
(a)  <prog>  ::=  <ifs>  |  <ifs> <elses>                (0), (1)

(b)  <ifs>  ::=  if( <vars> <equals> <vars> ){ <prog> }     (0)
             |   if( <vars> <equals> <vars> ){ <action> }   (1)

(c)  <elses>  ::=  else{ <action> }  |  else{ <prog> }     (0), (1)

(d)  <action>  ::=  goto(nearestPill)                       (0)
                |   goto(nearestPowerPill)                  (1)
                |   goto(nearestEdibleGhost)                (2)

(e)  <equals>  ::=  <  |  <=  |  >                       (0), (1), (2)
                |  >=  |  ==                               (3), (4)

(f)  <vars>  ::=  thresholdDistanceGhost                    (0)
              |   inedibleGhostDistance                     (1)
              |   avgDistBetGhosts  |  windowSize          (2), (3)
```

To better understand how the genotype-phenotype mapping process works in GE, here is a brief example. Suppose that we use the grammar defined previously. It is easy to see that each rule has a number of different choices. That is, there are 2, 2, 3, 5, and 4 choices for rules (a), (b), (c), (d), (e), and (f), respectively. Given the following genome: `16 93 34 81 17 46`, we need to define a mapping function (i.e., genotype-phenotype mapping) to produce the phenotype. GE uses the following function: $Rule = \mathrm{c}\ mod\ \mathrm{r}$, where c is the codon integer value and r is the number of choices for the current symbol, to determine which productions are picked for the phenotype. Beginning with the start symbol, `<prog>` , and its definition, `<prog> ::= <ifs> | <ifs> <elses>` the mapping function is performed: $16\ mod\ 2 = 0$. This means the left-most non-terminal, `<prog>` will be replaced by its 0^{th} production, `<ifs>` , leaving the current phenotype: `<ifs>` .

Because `<ifs>` has two productions and the next codon in the integer array is, 93, `<ifs>` is replaced by: `if( <vars> <equals> <var> ){ <action> }` . Following the same idea, we take the next codon, 34, and left-most non-terminal, `<vars>` and apply the mapping function. The results is 2, so the phenotype is now: `if( avgDistBetGhosts <equals> <var> ) { <action> }`. Repeating the same process for the remaining codons, we have the following expression: `if( avgDistBetGhosts <= inedibleGhostDistance ){goto(nearestPowerPill) }`. It is worth mentioning that in this example, all the codons were used. However, cases may occur where some codons are not used or, during the genotype-phenotype mapping, the end of the genome is reached and there are non-terminals remaining in the phenotype, causing it to be marked as invalid. If

this is the case, there are some options that one can use. For instance, the wrapping operator uses the idea that if a phenotype is incomplete, then the process continues starting from the first codon (from left to right) until a valid phenotype is built or the maximum number of wraps has been reached. If the phenotype is still incomplete at the end of this process, it will be necessary to assign the lowest possible fitness to the individual. As in GP, GE also uses crossover and mutation. The typical form of applying crossover in GE is selecting two genomes and randomly picking a crossover point on each of them. All codons beyond these points are swapped between the genomes. When applying a mutation, it is only necessary to select one genome and then replace a codon at random. It is also possible to direct the search operators like crossover and mutation towards the derivation trees generated during the genotype-phenotype mapping process, and thus operate as per standard GP. In this study genetic operators are applied at the genome level.

3 Our GE approach to Ms. Pac-Man

As highlighted by the literature there are many approaches one could take when designing a controller for Ms. Pac-Man. We now describe the rule-based approach we've taken. Broadly speaking, a rule is a sentence of the form "if <*condition*> then perform <*action*>". These rules are easy to read, understand, and more importantly, they can be combined to represent complex behaviours.

A number of functions were implemented to be used as primitives in the evolution of the Ms. Pac-Man controller (see Table 1). The aim of each of these functions is to be sufficiently basic, allowing evolution to combine them in a significant manner to produce the best possible behavior for the Ms. Pac-Man controller. In other words, we provide hand-coded, high-level functions and evolve the combination of these functions, pre-defined variables, and conditional statements using GE. These functions were easy to implement, and can be potentially very useful for our purposes. It is worth pointing out that we do not consider these functions to be optimal. For instance, in the case of the `AvoidNearestGhost()` function, we used a *window* that can provide some useful information to Ms. Pac-Man regarding the location of a potential dangerous ghost, but we could have also considered the idea of trying to guess the next position of a ghost given its current location and direction, or keeping track of all available paths in the entire maze given the location of the ghosts. It is also important to mention that these functions are not exclusive. That is, suppose when Ms. Pac-Man has eaten a power pill and is after a ghost, it may take a path full of pills or it can take a path that contains power pills. The latter is not an optimum scenario because it reduces significantly the chances of achieving the highest score possible.

3.1 Hand-Coded Example

The code shown in Fig. 1 calls the functions described in Table 1. It is worth mentioning that we tried different rule combinations with different values for the

Table 1. High-level functions used to control Ms. Pac-Man.

Function	*Variable*	*Description*
`NearestPill()`	`npd`	In the original version of this function [5] the agent finds the nearest food pill and heads straight for it regardless of what ghosts are in front of it. We modified it so that in the event a power pill is found before the target food pill, it waits next to the power pill until a different condition is met.
`NearestPowerPill()`	`nppd`	The goal of this function is to go to the nearest power pill.
`EatNearestGhost()`	`ngd`	When there is at least one edible ghost in the maze, Ms. Pac-Man goes towards the nearest edible ghost.
`AvoidNearestGhost()`	`ang`	Calculates the distance of the nearest inedible ghost in a "window" of size $windowSize \times windowSize$, given as a parameter set by evolution, and returns the location of the farthest node from the ghost. This "window" is a mask, where Ms. Pac-Man is at the center.
`NearestInedibleGhost()`	`nig`	Returns the distance from the agent to the nearest inedible ghost. This function is used by the previously explained `AvoidNearestGhost()`.

variables (e.g., windowSize) and the code shown in Fig. 1 gave us the highest score among all the combinations and different values assigned to the variable that we tested. First, we count the number of edible ghosts. Based on this information, Ms. Pac-Man has to decide if it goes to eat power pills, pills, or edible ghosts. We will further explain this hand-coded agent in Sect. 5 where we will compare it with the evolved controller. In the following section, the experimental setup is described to show how GE evolved the combination of the high-level functions described in Table 1.

4 Experimental Setup

We use Lucas' Ms. Pacman simulator [7]. It is important to mention that the simulator only gives one life to Ms. Pac-Man and has only one level. The Ms. Pac-Man implementation was tied into GE in Java (GEVA) [9][1]. This involved creating a grammar that is able to represent what was considered the best possible combination of the high level functions described in Table 1. This grammar can be seen in Fig. 2. The fitness function is defined to reward higher scores. This is done by adding the scores for each pill, power pill, and ghost eaten.

[1] Available from http://ncra.ucd.ie/geva

```
// edibleGhost counts for the number of edible ghosts.
windowSize = 13; avoidGhostDistance = 7; thresholdGhostDistanceGhosts = 10;
inedibleGhostDistance = Utilities.getClosest(current.adj, nig.closest, gs.getMaze());
switch(edibleGhosts){
 case 0:{
   if ( inedibleGhostDistance < windowSize ){
     next = Utilities.getClosest(current.adj, ang.closest, gs.getMaze());
   } else if ( numPowerPills > 0 ) {
     if ( avgDistBetGhosts < thresholdDistanceGhosts ){
       next = Utilities.getClosest(current.adj, nppd.closest, gs.getMaze());
     } else {
         next = Utilities.getClosest(current.adj, npd.closest, gs.getMaze());}

  } else { next = Utilities.getClosest(current.adj, npd.closest, gs.getMaze());}
    break;
  }
  case 1: case 2: case 3: case 4:{
    if ( inedibleGhostDistance < avoidGhostDistance ) {
       next = Utilities.getClosest(current.adj, ang.closest, gs.getMaze());
    }else {
       next = Utilities.getClosest(current.adj, ngd.closest, gs.getMaze()); }
    break;
  }
}
```

Fig. 1. Hand-coded functions to maneuver Ms. Pac-Man.

The experiments were conducted using a generational approach, a population size of 100 individuals, the ramped half and half initialisation method, and the maximum derivation tree depth, to control bloat, was set at 10. The rest of the parameters are as follows: tournament selection of size 2, int-flip mutation with probability 0.1, one-point crossover with probability 0.7, and 3 maximum wraps were allowed to "fix" invalid individuals (in case they still are invalid individuals, they were given low fitness values). To obtain meaningful results, we performed 100 independent runs. Runs were stopped when the maximum number of generations was reached.

5 Results and Discussion

5.1 The best evolved controller

The best individual found by GE (Fig. 3) is quite different from the hand-coded agent (Fig. 1). The first thing to notice are the differences in the values of the variables used in the conditional statements. For instance, `windowSize`, which is used by the function `AvoidNearestGhost()` has a different value. When we hand-coded our functions, we set the value at 13, whereas the evolved code set it at 19. Analysing these values, we can see that GE uses a wider window, so Ms. Pac-Man can have more information about the location of ghosts.

Let us continue examining the evolved code. The first condition, `if (edibleGhosts == 0)`, asks if all the ghosts are in an inedible state (in this state Ms. Pac-Man is unable to eat them) if so, it asks if there are power pills available (`numberPowerPills>0`). If this condition holds true, then it executes

```
<prog> ::= <setup><main>
<setup> ::= thresholdDistanceGhosts = <ghostThreshold>; windowSize = <window>;
            avoidGhostDistance = <avoidDistance>; avgDistBetGhosts = (int)adbg.score(gs);
            ang.score(gs, current, windowSize);
<main> ::= if(edibleGhosts == 0){ <statements> } else{ <statements> }
<statements> ::= <ifs> | <ifs> <elses>
<ifs> ::= if( <condition> ) { <action> } | if( <condition> ) { <statements> }
        | if( avgDistBetGhosts <lessX2> thresholdDistanceGhosts ) { <actsOrStats> }
        | if( inedibleGhostDistance <lessX2> windowSize) { <avoidOrPPill> }
<elses> ::= else { <action> } |  else { <statements> }
<actsOrStats> ::= <action> | <statements>
<action> ::= next = getClosest(current.adj, <closest>, gs.getMaze());
           | if ( numPowerPills <more> 0){ <pPillAction> }
             else{ next = getClosest(current.adj, npd.closest, gs.getMaze()); }
<closest> ::= npd.closest | ang.closest | ngd.closest
<avoidOrPPill> ::= <avoidAction> | <pPillAction>
<avoidAction> ::= next = getClosest(current.adj, <avoidClosest>, gs.getMaze());
<pPillAction> ::= next = getClosest(current.adj, <pPillClosest>, gs.getMaze());
<avoidClosest> ::= ang.closest
<pPillClosest> ::= nppd.closest
<condition> ::= <var> <comparison> <var>
<var> ::= thresholdDistanceGhosts | inedibleGhostDistance | avgDistBetGhosts
        | avoidGhostDistance | windowSize
<ghostThreshold> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
                   | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20
<avoidDistance> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15
<window> ::= 3 | 5 | 7 | 9 | 11 | 13 | 15 | 17 | 19
<comparison> ::= <less> | <more> | <lessE> | <moreE> | <equals>
<lessX2> ::= <less> | <lessE>
<less> ::= "<"
<more> ::= ">"
<lessE> ::= "<="
<moreE> ::= ">="
<equals> ::= "=="
```

Fig. 2. The grammar used in our experiments to evolve a Ms. Pac-Man controller using the functions described in Table 1.

the `NearestPowerPill()` method. This is quite an interesting sequence of conditions/instructions because it tries to rapidly increase Ms. Pac-Man's score by eating a power pill and then heading to the nearest edible ghost (shown in the second part of conditions/instruction). It is worth noting that this is very different from the previously used, hand-coded approach (see Fig. 1), where it takes a more conservative approach.

If we carefully analyse the last part of the evolved controller (see Fig. 3), we can see that only the last instruction is executed (`next=Utilities.getClosest(current.adj, ngd.closest, gs.getMaze());`). This is because there is a condition that is never met: `if (thresholdDistanceGhosts <= windowSize)`. There is also another element worth mentioning. The function `NearestInedibleGhost()` is never called by the evolved agent. These two elements are absent from the evolved controller indicating that evolution ignored them for the purpose of maneuvering Ms. Pac-Man through the maze. The evolved controller, however, achieved that highest score among all the Ms. Pac-Man agents, as shown in Table 2.

```
thresholdDistanceGhosts = 20; windowSize = 19; avoidGhostDistance = 14; avgDistBetGhosts =
(int) adbg.score(gs, thresholdDistanceGhosts);ang.score(gs, current, windowSize);
if (edibleGhosts == 0) { if (numPowerPills > 0) {
          next = Utilities.getClosest(current.adj, nppd.closest, gs.getMaze()); }
}else { if (thresholdDistanceGhosts <= windowSize) {
          next = Utilities.getClosest(current.adj, ang.closest, gs.getMaze()); }
       else {
          next = Utilities.getClosest(current.adj, ngd.closest, gs.getMaze()); } }
```

Fig. 3. Evolved controller used to guide Ms. Pac-Man.

5.2 Benchmarking Performance

In addition to the hand-coded agent and the evolved agent, we used three other Ms. Pac-Man agents (implemented in the code developed by [7]) for comparison purposes. The *Random* agent chooses one of five options (up, down, left, right, and neutral) at every time step. This agent allows reversing at any time. The second agent, called *Random Non-Reverse*, is the same as the random agent except it does not allow Ms. Pac-Man to back-track her steps. Finally, the *Simple Pill Eater* agent heads for the nearest pill, regardless of what is in front of it.

To compare all five different Ms. Pac-Man agents, three ghost teams already implemented in [7] were used. The random ghost team chooses a random direction for each of the four ghosts every time the method is called. This method does not allow the ghosts to reverse. The second team, Legacy, uses four different methods, one per ghost. Three ghosts use the following distance metrics: Manhattan, Euclidean, and a shortest path distance. Each of these distance measures returns the shortest distance to Ms. Pac-Man. The fourth ghost simply makes random moves. Finally, the Pincer team aims to trap Ms. Pac-Man between junctions in the maze paths. Each ghost attempts to pick the closest junction to Ms. Pac-Man within a certain distance in order to trap her.

In Table 2, we show the results for the five different Ms. Pac-Man agents vs. the three different ghost teams, described in the previous paragraph. As expected, the results achieved by these agents versus ghosts are poor. This is not surprising given their nature. It is very difficult to imagine how a controller that does not take into account any valuable information in terms of both, surviving and maximizing the score, can successfully navigate the maze. There are, however, some differences worth mentioning. For instance, random agent shows the poorest performance of all the agents explained previously. This is to be expected mainly because of two reasons: it performs random movements and, more importantly, it allows reversing at any time, so Ms. Pac-Man can easily spend too much time going backwards and forwards in a small space. This is different for the random non-reverse agent that does not allow reversing and as a result of this achieves a higher score. The score achieved by the simple pill eater is better compared with random and random non-reverse agents. This is simply because there is a target of increasing the score by eating pills.

Now, let us take a look at the last two controllers: hand-coded and evolved. The former was designed by the authors in order to achieve the highest score

Table 2. Results of the five different Ms. Pac-Man agents vs. three different ghost teams over 100 independent runs. Highest scores are shown in boldface.

Ghost Team	*Minimum Score*	*Maximum Score*	*Standard Deviation*	*Sum of all Runs*
	Random Agent			
Random Team	70	810	160.95	24,450
Legacy Team	40	200	31.75	8,670
Pincer Team	40	410	4.33	10,460
	Random Non-Reverse Agent			
Random Team	80	2,800	59.92	89,760
Legacy Team	80	5,310	74.40	69,950
Pincer Team	80	3,810	74.19	73,510
	Simple Pill Eater Agent			
Random Team	240	4,180	108.70	146,010
Legacy Team	250	5,380	107.04	154,720
Pincer Team	240	4,780	96.33	174,370
	Hand-coded Agent			
Random Team	180	11,220	242.68	579,590
Legacy Team	190	11,740	236.58	404,640
Pincer Team	790	12,820	327.10	409,040
	Evolved Agent			
Random Team	480	**11,640**	274.94	428,860
Legacy Team	470	**12,350**	311.60	394,560
Pincer Team	470	**13,830**	405.07	636,180

possible. This was done by eating a power pill (if all the ghost are inedible) and then heading straight to the nearest edible ghosts while avoiding inedible ghosts. Once a ghost has been eaten by Ms. Pac-Man, it returns to the ghost nest, resets its status to inedible, and re-enters the maze. The big difference between the hand-coded controller (depicted in Fig. 1) and the evolved controller (shown in Fig. 3) is that the latter takes a more risk-based approach by heading for the power pill (each of these awards 50 points) and then heading for edible ghosts (without taking into account if there are inedible ghosts in the way of Ms. Pac-Man), whereas the former takes a more conservative approach by taking into account the positions of pontential dangerous ghosts and if any of these are in the path of Ms. Pac-Man, it tries to avoid the ghost(s). As can be seen in Table 2 the highest score, regardless of the ghost team used, was achieved by the evolved controller.

6 Conclusions

This work proposes a method to evolve high-level functions, described in Table 1, to maneuver Ms. Pac-Man through a maze where the goal is to achieve the highest possible score while avoiding dangerous ghosts. To achieve this goal we

used GE for its flexibility in specifying rules in the form of "if <*condition*> then perform <*action*>". These rules were combined by means of evolution and the resulting evolved controller (Fig. 3) achieved the highest score (Table 2) compared against four other controllers, including a hand-coded controller. All competitors were played against three different ghost teams also described above. As can be seen, the evolved controller is different from the hand-coded controller (shown in Fig. 1) in the sense that the former takes a more risk-based approach whereas the latter is more conservative by checking the positions of ghosts. It is also important to note that the evolved controllers here did not match or exceed the score of Matsumoto's [6] (he used a hand-coded agent). However, this is not discouraging due to the fact that our controller was only allowed one level and one life where Matsumoto's was given three initial lives, could earn more lives, and had more than one level to play.

Acknowledgments

This research is based upon works supported by the Science Foundation Ireland under Grant No. 08/IN.1/I1868.

References

1. I. Dempsey, M. O'Neill, and A. Brabazon. *Foundations in Grammatical Evolution for Dynamic Environments*. Springer, Apr. 2009.
2. M. Gallagher. Learning to play pac-man: An evolutionary, rule-based approach. In *In CEC 03, The 2003 Congress on Evolutionary Computation*, pages 2462–2469. IEEE, 2003.
3. J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, Massachusetts, 1992.
4. S. Lucas. Evolving a neural network location evaluator to play ms. pac-man. In *IEEE Symposium on Computational Intelligence and Games*, pages 203–210, 2005.
5. S. Lucas. Ms Pac-Man Competition. http://cswww.essex.ac.uk/staff/sml/pacman/PacManContest.html, September 2009.
6. S. Lucas. Ms Pac-Man Competition - IEEE CIG 2009. http://cswww.essex.ac.uk/staff/sml/pacman/CIG2009Results.html, September 2009.
7. S. Lucas. Ms Pac-Man versus Ghost-Team Competition. http://csee.essex.ac.uk/staff/sml/pacman/kit/AgentVersusGhosts.html, September 2009.
8. M. O'Neill and C. Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*. Kluwer Academic Publishers, 2003.
9. M. ONeill, E. Hemberg, C. Gilligan, E. Bartley, J. McDermott, and A. Brabazon. GEVA - grammatical evolution in java (v 1.0). Technical report, UCD School of Computer Science, 2008.
10. I. Szita and A. Lõrincz. Learning to play using low-complexity rule-based policies: illustrations through ms. pac-man. *J. Artif. Int. Res.*, 30(1):659–684, 2007.