Comparing the Performance of the Evolvable πGrammatical Evolution Genotype-Phenotype Map to Grammatical Evolution in the Dynamic Ms. Pac-Man Environment

Edgar Galván-López, David Fagan, Eoin Murphy, John Mark Swafford,
Alexandros Agapitos, Michael O'Neill and Anthony Brabazon

***Abstract*— In this work, we examine the capabilities of two forms of mappings by means of Grammatical Evolution (GE) to successfully generate controllers by combining high-level functions in a dynamic environment. In this work we adopted the Ms. Pac-Man game as a benchmark test bed. We show that the standard GE mapping and Position Independent GE (πGE) mapping achieve similar performance in terms of maximising the score. We also show that the controllers produced by both approaches have an overall better performance in terms of maximising the score compared to a hand-coded agent. There are, however, significant differences in the controllers produced by these two approaches: standard GE produces more controllers with invalid code, whereas the opposite is seen with πGE.**

I. Introduction

In Grammatical Evolution (GE) [10], [1], rather than representing programs as parse trees, as in Genetic Programming (GP) [4], a variable length linear genome representation is used. A genotype to phenotype mapping process is employed on these genomes which uses a user-specified context-free grammar to generate the actual phenotype. This work is concerned with understanding the impact of two forms of this mapping, the traditional GE mapping and Position Independent GE (πGE) [9], in a dynamic environment. We use the notion of a dynamic problem as defined in [1] *"... a problem in which some element under its domain varies with the progression of time"*. For this purpose, we use the Ms. Pac-Man game as a benchmark problem (the specifics about the Ms. Pac-Man game are given in Section III).

Using Trojanowski and Michalewicz's categorization of dynamic problems [12], [13], we know that this problem lies in the category of a static objective function (i.e., maximizing the score of the Ms. Pac-Man agent) and static constraints. These type of problems are interesting because both elements (objective function and constraints) do not change over time and, in principle, it should be easier to examine the effects of both mappings (i.e., standard and and πGE) on a dynamic environment.

GE has been successfully used in a wide range of applications as reported in [10], [1]. πGE has been reported to have a better overall performance in terms of finding better results compared to standard GE as shown in [9], [2]. The objective of this paper is to see the utility of both forms of mappings on a dynamic problem.

This paper is structured as follows. In the following section we describe how both mappings (standard GE and πGE) work. In Section III we describe the benchmark problem used in this work. In Section IV we describe the approach taken for controlling our agent on a dynamic environment. In Section V we describe the experimental setup and Section VI presents the results achieved by our approach, followed by a discussion. Finally, Section VII draws some conclusions.

Edgar Galván-López, David Fagan, Eoin Murphy, John Mark Swafford, Alexandros Agapitos, Michael O'Neill and Anthony Brabazon are with the University College Dublin, Natural Computing Research & Applications Group, UCD CASL, 8 Belfield Office Park, Beaver Row, Clonskeagh, Dublin 4, email: edgar.galvan, david.fagan, eoin.murphy, john-mark.swafford, alex-agapitos, m.oneill, anthony.brabazon@ucd.ie.

II. Overview of GE and πGE Mappings

A. Standard GE Mapping

In GE, 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 Figure 4 for the actual grammar used in our studies):

```
   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, 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,

978-1-4244-8126-2/10/$26.00 ©2010 IEEE

we need to define a mapping function (i.e., genotype-phenotype mapping) to produce the phenotype. GE uses the following function: $Rule$ = c mod 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 will have the following expression:
`if( avgDistBetGhosts <= inedibleGhostDistance ) {goto(nearestPowerPill) }`.

B. πGE Mapping

Position Independent GE (πGE) [9], [2] is almost identical in operation to standard GE except when it comes to the mapping process. As shown above, GE always takes the left-most non-terminal to be expanded next during the mapping from genotype to phenotype. In πGE, we refine this process by introducing an evolved ordering as to which non-terminals are expanded. By doing this, we allow the grammar to have more freedom in the way the derivation tree (i.e., phenotype) is built. With this method, not only the productions used for the non-terminal expansions are picked using chromosome, the chromosome is also used to determine the order in which the non-terminals are expanded. In πGE, we take a standard GE chromosome and convert it into a list of codon pairs. The second codon of each pair is used to choose which non-terminal to expand next (the order of mapping), and the first codon is used to decide what that non-terminal will expand to. If this happens to be another non-terminal it is added to a list of unexpanded non-terminals and is available to be selected for further expansion. The method for selection which non-terminal expand is based on the equation for selecting what the non-terminal should expand to: $non-terminal\ to\ expand = codon\ value\ mod\ number\ of\ non-terminals$.

C. Final Remarks on GE and πGE

As mentioned previously, the GE mapping follows a left-to-right, depth-first development of the structure, whereas πGE adopts an evolved ordering to the mapping. This mapping, is in fact, quite interesting, specially when dealing with the dynamic problem that we have used as benchmark test. This is introduced in the following section.

III. Ms. PacMan Agent

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, Blinky, Pinky, Inky, and Sue, 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 predictable. This feature makes the game extremely challenging.

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. 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 is 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. Due to their non-deterministic behaviour, the player may not be certain what the ghosts will do next. 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 [6]). The highest score achieved by a computer, developed by Matsumoto [7], based on a screen-capture system that is supposed to be exactly the same as the arcade game, stands at 30,010 [6]. The other top three scores achieved are 15640, 9000 and 8740 points, respectively [7]. It is worth pointing out that all of them used a hand-coded approach as opposed to an evolutionary computation or machine learning algorithm.

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 [4]. He used genetic programming to combine

pre-defined actions and conditional statements to evolve 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 that of Koza [4] is reported by Szita and Lõrincz [11]. 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. This approach is similar to Koza's in that they pre-define a set of conditions and actions and allow the Ms. Pac-Man agent to learn how to combine and prioritise them. Another, more recent, approach by Lucas [5] uses an evolutionary strategy to train a neural network to play Ms. Pac-Man in hopes of creating the best possible player. Recently, Galván-López et al. used GE [3].

IV. GE and πGE Approach to Ms. PacMan Agent

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 have 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. Now, an important question arises: What needs to be accounted for when using rules to evolve a Ms. Pac-Man controller? To answer this question it is necessary to define these three elements:

- *Conditions* - The current state of the ghosts, Ms. Pac-Man, the pills in the maze, and their relations to each other are used to define the conditions. It is important to consider these because the combinations of them will determine which actions Ms. Pac-Man will take to achieve high scores.
- *Actions* - Given the goal of maneuvering Ms. Pac-Man through a maze while trying to get the highest score possible, a set of basic actions need to be defined to determine how Ms. Pac-Man will move through the maze. When certain conditions, determined by evolution, are met these actions will be executed. Descriptions of the actions defined for this work can be found in Table I.
- *Complexity* - Because there are many possibilities for combining the actions and conditions, restrictions are needed to limit the number of these combinations. When defining these restrictions, a balance is needed to ensure that the evolved controllers may increase in complexity without becoming completely unreasonable in size. To achieve this, a grammar is defined which specifies what combinations of conditions and actions are possible (see Figure 4 for the grammar used).

A number of functions were implemented to be used as primitives in the evolution of the Ms. Pac-Man controller (see Table I). 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.

```
// 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 (see Table I for a full description of the functions used).

A. Hand-Coded Example

The code shown in Figure 1 calls the functions described in Table I. It is worth mentioning that we tried different rule combinations with different values for the variables (e.g., windowSize) and the code shown in Figure 1 gave us the highest score among all the combinations and different values assigned to the variable that we tested. As stated before, the goal of the game is to maneuver Ms. Pac-Man through a maze, trying to achieve the highest score possible while trying to avoid inedible ghosts. 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 Section VI where we will compare it with the evolved controllers (both using GE and πGE). In the following section, the experimental setup is described to show how both approaches evolved the combination of the high-level functions described in Table I with conditional statements and variables to determine when certain actions should be taken.

TABLE I

HIGH-LEVEL FUNCTIONS USED TO CONTROL MS. PAC-MAN.

Function	*Variable*	*Description*
`NearestPill()`	`npd`	Originally, this function [6] 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 and another action is executed.
`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 just 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()`.

```
// edibleGhost counts for the number of edible ghosts.
1 thresholdDistanceGhosts = 13; windowSize = 11 ;
2 avoidGhostDistance = 8 ;
3 avgDistBetGhosts = (int)adbg.score(gs,
4                                    thresholdDistanceGhosts);
5 ang.score(gs, current, windowSize);
6 if(edibleGhosts == 0){
7    if (avgDistBetGhosts >= avoidGhostDistance ) {
8       if (numPowerPills > 0){
9         next = Utilities.getClosest(current.adj,
10                            nppd.closest , gs.getMaze());
11      } else{
12         next = Utilities.getClosest(current.adj,
13                             npd.closest, gs.getMaze());
14      }
15   } else {
16     if( avgDistBetGhosts <= avgDistBetGhosts ) {
17       if (avoidGhostDistance>thresholdDistanceGhosts){
18           next = Utilities.getClosest(current.adj,
19                            ngd.closest , gs.getMaze());
20        }
21      } else {
22        if (numPowerPills > 0){
23          next = Utilities.getClosest(current.adj,
24                           nppd.closest , gs.getMaze());
25        } else{
26          next = Utilities.getClosest(current.adj,
27                            npd.closest, gs.getMaze());
28       }
29      }
30   }
31
32 } else{
33   if (inedibleGhostDistance < windowSize) {
34       next = Utilities.getClosest(current.adj,
35                           nppd.closest , gs.getMaze());
36    } else {
37     next = Utilities.getClosest(current.adj,
38                            ngd.closest , gs.getMaze());
39    }
40 }
```

Fig. 2. Best evolved agent using GE to maneuver Ms. Pac-Man (see Table I for a full description of the functions used).

```
// edibleGhost counts for the number of edible ghosts.
thresholdDistanceGhosts = 10; windowSize = 11 ;
avoidGhostDistance = 4;
avgDistBetGhosts = (int)adbg.score(gs,
                                    thresholdDistanceGhosts);
ang.score(gs, current, windowSize);
if (edibleGhosts == 0){
   if (inedibleGhostDistance < windowSize) {
      next = Utilities.getClosest(current.adj,
                               nppd.closest , gs.getMaze());
   }
 }
   else{
     if (inedibleGhostDistance < windowSize) {
        next = Utilities.getClosest(current.adj,
                               nppd.closest , gs.getMaze());
     } else {
        next = Utilities.getClosest(current.adj,
                                ngd.closest , gs.getMaze());
     }
}
```

Fig. 3. Best evolved agent using πGE to maneuver Ms. Pac-Man (see Table I for a full description of the functions used).

V. EXPERIMENTAL SETUP

We use the Ms. Pac-Man simulator developed by Simon Lucas [8]. 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). 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 I. The grammar used in this work is shown in Figure 4. The fitness function is defined to reward higher scores. This is done by adding the scores for each pill, power pill, and ghost eaten. The scores used are the same as the original Ms. Pac-Man game described in Section III. Each generated Ms. Pac-Man agent was executed 20 times to get the fitness.

The experiments were conducted using a generational approach, a population size of 100 individuals, 100 generations, 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 lowest possible fitness). To obtain meaningful results, we performed 100 independent runs. Runs were stopped when the maximum number of generations was reached.

VI. Results and Discussions

A. Ghost Teams and Basic Controllers

For comparison purposes, we used three different ghost teams (already implemented in [8]), called Random, Legacy, and Pincer team, where each has a particular form of "attacking" Ms. Pac-Man. 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.

We started our studies by using four different Ms. Pac-Man Agents (including a hand-coded approach as mentioned in Section IV). These are random, random non-reverse and simple pill eater agent. 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 backtrack her steps. Finally, the *Simple Pill Eater* agent heads for the nearest pill, regardless of what is in front of it. Results achieved by these agents are shown in Table II.

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, the random agent shows the poorest performance of all the agents described 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.

TABLE II

Results of four *different Ms. Pac-Man agents (random, random non-reverse, simple pill eater and a hand-coded agent)* vs. three different ghost teams over 100 independent runs. Highest scores are shown in boldface.

Ghost Team	*Min. Score*	*Max. Score ± Std. Dev.*	*Sum of all Runs*
	Random Agent		
Random	70	810 ± 160.95	24,450
Legacy	40	200 ± 31.75	8,670
Pincer	40	410 ± 4.33	10,460
	Random Non-Reverse Agent		
Random	80	2,800 ± 59.92	89,760
Legacy	80	5,310 ± 74.40	69,950
Pincer	80	3,810 ± 74.19	73,510
	Simple Pill Eater Agent		
Random	240	4,180 ± 108.70	146,010
Legacy	250	5,380 ± 107.04	154,720
Pincer	240	4,780 ± 96.33	174,370
	Hand-coded Agent		
Random	180	**11,220** ± 242.68	579,590
Legacy	190	**11,740** ± 236.58	404,640
Pincer	790	**12,820** ± 327.10	409,040

B. Evolved Controllers

Due to space limitations, we have taken the best and the worst four individuals from the 100 independent runs using both GE and πGE (i.e., 16 individuals). Each of these were used 100 times in the Ms. Pac-Man game. We can see the highest scores achieved by the best controllers using GE and πGE in Tables III and IV, respectively. Clearly, the best four controllers evolved using GE show a better overall performance in terms of highest score (i.e., 8 results were better compared to the hand-coded controller). πGE also shows good performance, although not as good as GE, as can be seen in Table IV (i.e., 5 maximum scores were better compared to the hand-coded controller shown in Figure 1).

To see how robust our approach is, we now are turning our attention to the results achieved by the "worst" evolved controllers using both approaches. Table V shows the results obtained by the "worst" four evolved controllers obtained by GE on 100 games. The maximum scores achieved by these evolved controllers show that they are robust in achieving a high score (i.e., 6 out of 12 are at least as good as the ones found by the hand-coded approach). A similar story is observed by the "worst" evolved controllers found by πGE (see Table VI), where 7 maximum scores were higher than those found by the hand-coded approach. Figure 5 simply plots the highest scores found by the best and the "worst" controllers found by GE and πGE.

To understand how the evolved controllers manage to achieve highest scores compared to the hand-coded approach, it is necessary to analyse the controllers. Due to space constraints, we will use the best controller found by GE (see Figure 2) and πGE (see Figure 3) and compared them with the hand-coded controller (shown in Figure 1). If we start first analysing our hand-coded controller, we can see that we take quite a conservative approach. For instance, notice how we considered that *AvoidNearestGhost()* function is important because it helps Ms. Pacman to eventually scape from the ghosts. However, this function is never called from the evolved controllers (both using GE and πGE). In fact, the latter two controllers both use a more risky approach. That is, they achieved the highest scores compared to the hand-coded approach by using only three functions: *NearestPowerPill(), EatNearestGhost(), NearestPill()*. This is

```
<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. 4. The grammar used in our experiments to evolve a Ms. Pac-Man controller using the functions described in Table I.

TABLE III

RESULTS OF THE FOUR *fittest evolved Ms. Pac-Man agents using Standard GE* VS. THREE DIFFERENT GHOST TEAMS OVER 100 INDEPENDENT RUNS. HIGHEST SCORES ARE SHOWN IN BOLDFACE .

Ghost Team	*Min. Score*	*Max. Score ± Std. Dev.*	*Sum of all Runs*
	Evolved Agent 1		
Random	230	11010 ± 2704.86	356450
Legacy	260	**13780** ± 2666.00	418380
Pincer	310	**14180** ± 3397.97	417500
	Evolved Agent 2		
Random	570	**11220** ± 2290.83	441550
Legacy	660	12010 ± 3052.81	503790
Pincer	260	14010 ± 3606.44	663710
	Evolved Agent 3		
Random	580	9590 ± 2424.38	405410
Legacy	570	12110 ± 2879.62	484820
Pincer	1000	14050 ± 3556.82	630850
	Evolved Agent 4		
Random	230	10410 ± 2745.91	375120
Legacy	260	12000 ± 2651.73	467990
Pincer	900	12230 ± 2975.72	419880

TABLE IV

RESULTS OF THE FOUR *fittest evolved Ms. Pac-Man agents using πGE* VS. THREE DIFFERENT GHOST TEAMS OVER 100 INDEPENDENT RUNS. HIGHEST SCORES ARE SHOWN IN BOLDFACE.

Ghost Team	*Min. Score*	*Max. Score ± Std. Dev.*	*Sum of all Runs*
	Evolved Agent 1		
Random Team	230	**12850** ± 2717.12	350460
Legacy Team	340	10450 ± 2571.18	476080
Pincer Team	700	**13730** ± 3199.74	451760
	Evolved Agent 2		
Random Team	1040	9510 ± 2030.77	434910
Legacy Team	1080	**12390** ± 2591.76	530930
Pincer Team	280	10990 ± 2979.73	380590
	Evolved Agent 3		
Random Team	600	8800 ± 2248.27	412890
Legacy Team	1000	11650 ± 2733.71	462880
Pincer Team	270	12280 ± 3002.68	501470
	Evolved Agent 4		
Random Team	570	10980 ± 2528.87	444390
Legacy Team	270	11860 ± 3216.99	487090
Pincer Team	290	13230 ± 3870.62	470780

actually quite interesting because GE and πGE shape the controllers based on what gives the highest points (i.e., power pills and then heading towards the ghosts). This shows some degree of intelligence because this happens only if the four ghosts are in inedible state, so this gives the chance to Ms. Pacman to eat the four ghosts after eaten the power pill, and, so, trying to maximise the score.

There is one element that is different from the controllers evolved by GE and πGE. The former produced code that will never be executed because the conditions are never met (see Figure 2 lines 15-30). This is not the case for πGE (see Figure 3), where all the code can be executed at some point. It is important to point out that for both approaches the same grammar was used (see Figure 4). This is something that attracted our attention, and so, we examined all the controllers produced by both GE and πGE. We found out that GE produced 16.66% controllers that contain unused code, whereas only 3.33% of the evolved controllers produced by πGE contain unused code. When we analyse how this could happen, we see that *pi*GE produces bigger derivation trees

TABLE V

RESULTS OF THE FOUR *less-fit evolved Ms. Pac-Man agents using Standard GE* VS. THREE DIFFERENT GHOST TEAMS OVER 100 INDEPENDENT RUNS. HIGHEST SCORES ARE SHOWN IN BOLDFACE.

Ghost Team	*Min. Score*	*Max. Score ± Std. Dev.*	*Sum of all Runs*
	Evolved Agent 1		
Random	230	10180 ± 2341.41	361950
Legacy	340	10400 ± 2492.30	414410
Pincer	330	14290 ± 3024.45	451810
	Evolved Agent 2		
Random	870	10710 ± 2724.08	479250
Legacy	470	11390 ± 2996.76	395720
Pincer	1050	14010 ± 3712.82	771480
	Evolved Agent 3		
Random	470	**12820** ± 2866.94	523040
Legacy	470	**12940** ± 2943.21	399110
Pincer	530	**14420** ± 4456.26	809430
	Evolved Agent 4		
Random	230	10320 ± 2218.13	364970
Legacy	340	12240 ± 2661.62	466990
Pincer	450	10830 ± 2453.71	380030

TABLE VI

RESULTS OF THE FOUR *less-fit evolved Ms. Pac-Man agents using πGE* VS. THREE DIFFERENT GHOST TEAMS OVER 100 INDEPENDENT RUNS. HIGHEST SCORES ARE SHOWN IN BOLDFACE.

Ghost Team	*Min. Score*	*Max. Score ± Std. Dev.*	*Sum of all Runs*
	Evolved Agent 1		
Random	470	9610 ± 2464.54	421580
Legacy	870	10280 ± 2655.65	367270
Pincer	1670	13630 ± 3369.96	551100
	Evolved Agent 2		
Random	230	**11840** ± 2853.38	349030
Legacy	340	10740 ± 2690.25	495140
Pincer	330	13580 ± 3046.06	438250
	Evolved Agent 3		
Random	330	9720 ± 2727.93	411140
Legacy	290	**14090** ± 2863.72	432560
Pincer	430	13620 ± 2845.80	422880
	Evolved Agent 4		
Random	230	8820 ± 2417.84	333380
Legacy	330	11930 ± 2498.56	488500
Pincer	330	**14150** ± 3506.20	454930

(both in terms of depth and number of nodes) compared to those produced by GE (see Figure 5). This is not surprising because πGE's nature allows more freedom in the production of derivation trees compared to standard GE.

VII. CONCLUSIONS AND FUTURE WORK

We have shown that it is possible to successfully combine high-level functions by means of evolution using two forms of mappings: the traditional GE and πGE. Both approaches have a similar performance in terms of maximizing Ms. Pac-Man score. There are, however, important differences on these approaches. GE produces more controllers with invalid code (i.e., code that is never executed because a condition is never met), whereas with πGE we have a mirror image. That is, there are more controllers (five times compared to GE) where there is no invalid code. The former is not an ideal scenario because, as we have seen, both approaches give a similar performance (i.e., increasing the score), and so, πGE is a better approach when combining high-level functions.

We have also seen that our approach is robust in the sense that even those controllers having a low fitness achieved good results on the game, and in fact, some of them were as competitive as the best controllers found by GE and πGE. In a future work, we would like to explore the idea of using more complex functions and adopting a multi-objective approach, where both: surviving and maximizing score, can be taken into account to evolve our controller.

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] D. Fagan, M. O'Neill, E. Galván-López, A. Brabazon, and S. McGarraghy. An analysis of genotype-phenotype maps in grammatical evolution. In *EuroGP 2010: European Conference on Genetic Programming*. Springer, 2010.

[3] E. Galván-López, J. M. Swafford, M. O'Neill, and A. Brabazon. Evolving a ms. pacman controller using grammatical evolution. In *Applications of Evolutionary Computation, EvoApplications 2010: EvoCOMPLEX, EvoGAMES, EvoIASP, EvoINTELLIGENCE, EvoNUM and EvoSTOC*, volume 6024 of *LNCS*, pages 161–170. Springer, 2010.

[4] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, Massachusetts, 1992.

[5] S. Lucas. Evolving a neural network location evaluator to play ms. pacman. In *IEEE Symposium on Computational Intelligence and Games*, pages 203–210, 2005.

[6] S. Lucas. Ms Pac-Man Competition. `http://cswww.essex.ac.uk/staff/sml/pacman/PacManContest.html`, September 2009.

[7] S. Lucas. Ms Pac-Man Competition - IEEE CIG 2009. `http://cswww.essex.ac.uk/staff/sml/pacman/CIG2009Results.html`, September 2009.

[8] S. Lucas. Ms Pac-Man versus Ghost-Team Competition. `http://csee.essex.ac.uk/staff/sml/pacman/kit/AgentVersusGhosts.html`, September 2009.

[9] M. O'Neill, A. Brabazon, M. Nicolau, S. McGarraghy, and P. Keenan. pi-grammatical evolution. In K. Deb, R. Poli, W. Banzhaf, H.-G. Beyer, E. K. Burke, P. J. Darwen, D. Dasgupta, D. Floreano, J. A. Foster, M. Harman, O. Holland, P. L. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, and A. M. Tyrrell, editors, *GECCO (2)*, volume 3103 of *Lecture Notes in Computer Science*, pages 617–629. Springer, 2004.

[10] M. O'Neill and C. Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*. Kluwer Academic Publishers, 2003.

[11] 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.

[12] K. Trojanowski and Z. Michalewicz. Evolutionary algorithms for non-stationary environments. In *In Proc. of 8th Workshop: Intelligent Information systems*, pages 229–240. ICS PAS Press, 1999.

[13] K. Trojanowski and Z. Michalewicz. Evolutionary optimization in non-stationary environments. *Journal of Computer Science and Technology*, 1(2):93–124, 2000.

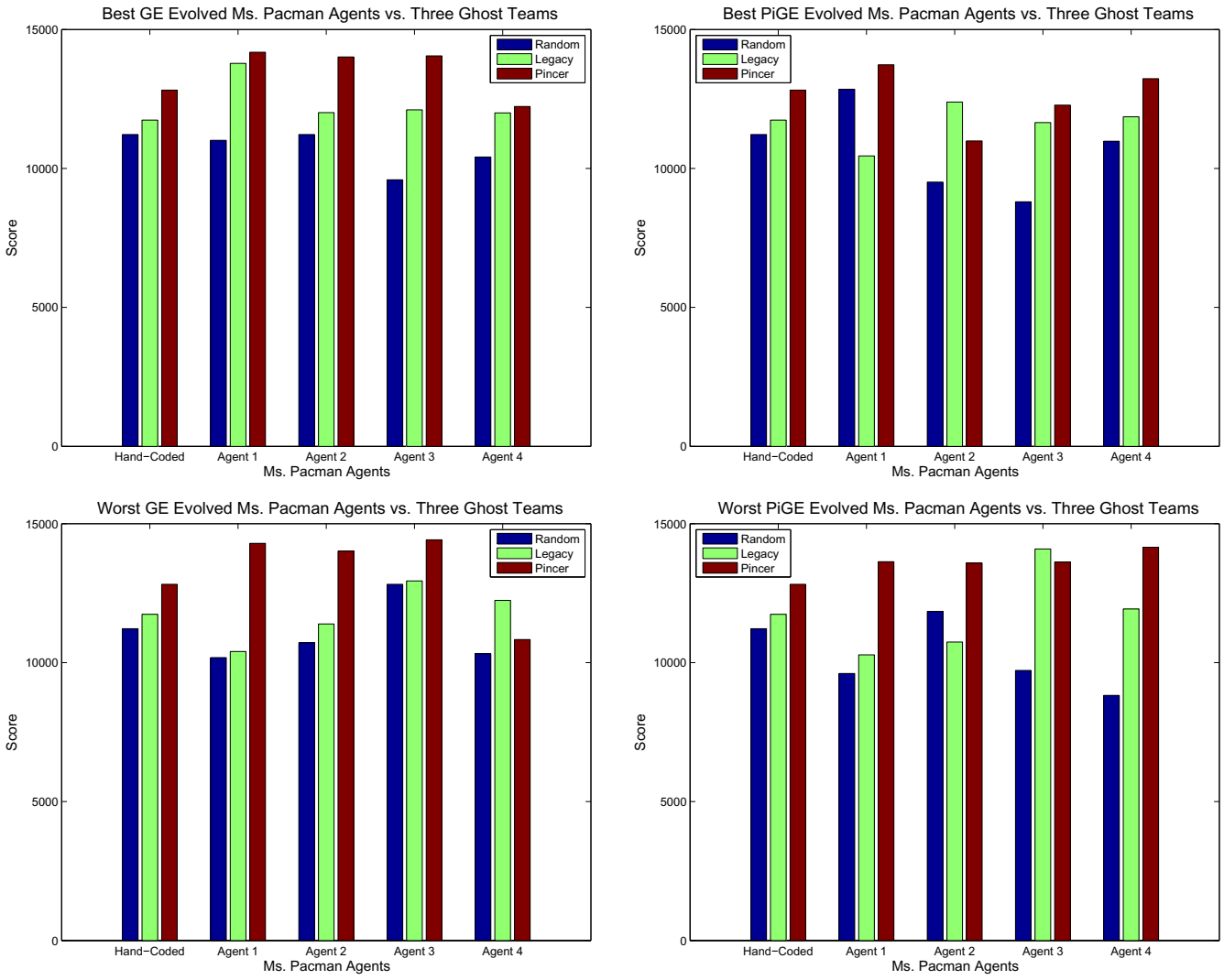

Fig. 5. Highest scores achieved by the best GE evolved controller (top left), best πGE evolved controller (top right), "worst" GE evolved controller (bottom left) and "worst" πGE evolved controller (bottom right).

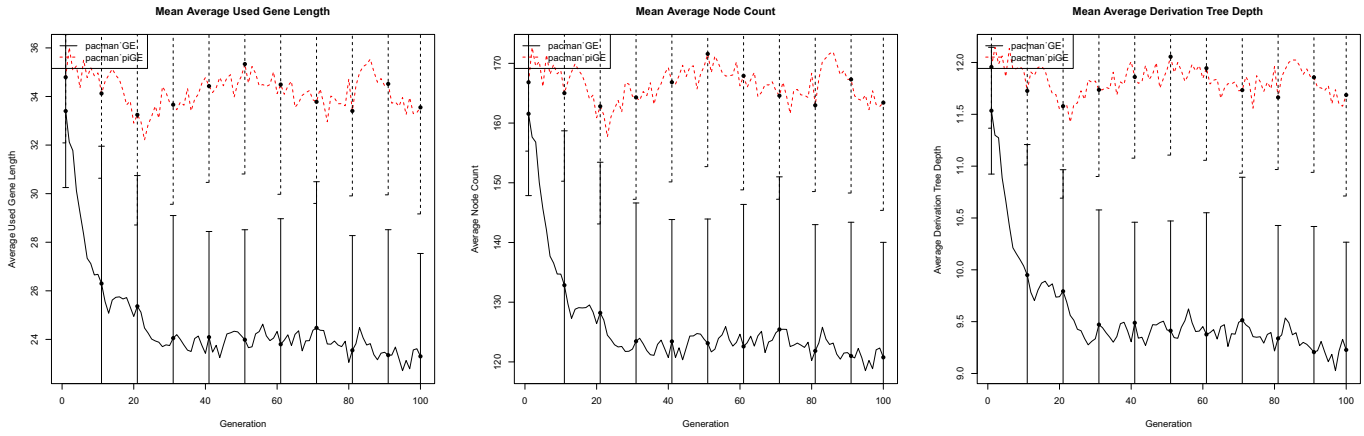

Fig. 6. Number of used genes (left), number of nodes in the derivation tree (centre) and depth of the derivation tree (right) for Standard GE and πGE on the Ms. Pac-Man problem.