A Pac-Man bot based on Grammatical Evolution*

Héctor Laria Mantecón, Jorge Sánchez Cremades, José Miguel Tajuelo Garrigós, Jorge Vieira Luna, Carlos Cervigon Rückauer, Antonio A. Sánchez-Ruiz

Dep. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid (Spain)
{hlaria, jorsan06, jtajuelo, jovieira, ccervigon, antsanch}@ucm.es

Abstract. In this article, we propose the development of a bot for playing the video game Ms. Pac-Man vs. Ghosts using a grammatical evolution based evolutionary algorithm. This technique evolves programs that are evaluated by executing them in the game. The program encodes the strategy that the bot plays and is obtained through the derivation of grammar rules in a particular order, which is defined by the algorithm. We experimented with two different grammars: The first one includes high-level actions and the second one involves medium-level actions. Both grammars include state providers. To make the evolutionary process more efficient, we perform a series of optimizations on the evolutionary algorithm, including parallelization of the fitness evaluation and multi-objective optimization. Experimental results using the two grammars and two different ghost controllers are presented. We report better results with our bots than the baseline controllers and other controllers based on grammatical evolution.

Keywords: Genetic programming, grammatical evolution, multi-objective optimization, decision trees, Pac-Man

1 Introduction

Ever since the birth of video-games we've seen artificial intelligence techniques applied to them: Character behaviour, enemy strategies, path-finding, etc. We want to explore Grammatical Evolution (a Genetic Programming variant) to evolve game strategies generated from the derivation of defined grammar rules. For this purpose, we experimented with the evolution of a bot for Ms. Pac-Man, a well-known game which can have many sub-goals, like surviving the most time possible, eating the most pills, killing as many ghosts as it can, or go through a lot of levels before dying to the ghosts.

* Supported by Spanish Ministry of Economy, Industry and Competitiveness under grants TIN2014-55006-R and TIN2014-57028-R

Particularly, we experimented with controllers based on two different grammars, with high and medium level actions respectively. Due to the complexity of video-games and how useful it could be for an artificial intelligence to modify its behaviour in real time, we want to check the results of multi-objective optimization in grammatical evolution, and how we can achieve the sub-goals we consider more important in a situation by simply changing the evaluation functions we use in the grammatical evolution algorithm.

We will show that this approach based on Grammatical Evolution gets excellent results and we will see that the bots produced can obtain high scores and complete several levels, even better results than the coded bots included, or other known evolutionary bots.

The rest of the article is structured as follows: Section 2 describes the techniques and the work we've found related to our project. Section 3 gives information on the Pac-Man framework we use and the bots we experiment with. Section 4 explain our bot in detail showing some results and comparatives. Section 5 describes multi-objective optimization and its results, and we will compare them to the previous ones. Finally, in section 6, we discuss the conclusions of this study and the future work.

2 Related Work

2.1 Genetic Programming

Genetic Programming (GP) [16] is one of the many different branches of algorithms that exist in the field of Evolutionary Algorithms. The goal in GP is to produce the best program, in a determined programming language, to solve a particular problem. The programs are usually encoded using a tree structure (genotype) in which each node represents a token of the chosen programming language. Certain individuals in the population will be selected, recombined and transformed slightly by different types of selection, crossover and mutation operators. In order to evaluate the performance of each individual, its genome will be transformed into the final program (phenotype) that will be executed in the context of the particular problem to solve. This process is repeated several times, and the final solution will be the individual representing the program with the best score of the whole population [3].

The main drawback of GP is that the use of trees to encode the genotype is very demanding in terms of memory, especially when bloating occurs [15], and that the selection, crossover and mutation operators are relatively slow because they work with recursive tree structures.

GP has been used previously to evolve AI controllers for the Pac-Man game. In particular, Koza [9] used a set of high level operators for maze information retrieval (*DISF-Distance to Fruit*) and Pac-Man direct control (*AFRUIT-Advance to Fruit*) to develop a bot for a custom version of the game. Alhejali and Lucas [2] also used GP with even more abstract operators (*isInDanger, IsToEnergizerSafe or toSafety*) to automatically evolve Pac-Man players. Brandstetter

And Ahmadi [5], on the other hand, used low level action operators for Pac-Man movement (*UP, DOWN...*), obtaining better results (score) compared with the previous controllers.

2.2 Grammatical Evolution

Grammatical Evolution (GE) [13] is similar to Genetic Programming in that both evolve programs to find the best one to solve a particular problem, but they differ on the encoding (genotype). While GP uses trees, GE uses an array of integers where each integer represents the rule of the grammar (in BNF) that will be chosen to produce the program (phenotype) when the genotype is decoded. That is, instead of storing the syntactic tree, in GE each individual stores information to derive the program from the language grammar. This way, the process of crossover and mutation works with integer arrays, it is much faster and consumes less memory. The algorithm is also independent of the domain, so we can solve different problems just changing the grammar to define the space of valid programs.

Although classical crossover operators, like the single-point crossover, and classical mutation operators, like integer flip mutation, can be used in GE algorithms, they tend to produce too much noise in the phenotype, especially crossover which generates chaotic populations. For this reason, new operators were been developed to try to avoid this destructive behaviour, like LHS replacement crossover, which tries to do the crossover less destructive by taking into account the phenotype structure and not just the genotype [7] and Neutral Mutation for improving population's diversity [12].

Grammatical Evolution (GE) has also been previously used to evolve Pac-Man controllers. Galván-López [6] used a similar approach to Koza's using high level operators for movement (*ANG - Avoid Nearest Ghost*) and information retrieval (*avgDistBetGhosts*). They used a grammar with if-else statements to achieve determined outputs based on some game conditions. With this approach, they achieved similar results to GP controllers with the advantage that the BNF could be changed easily adding restrictions or features easily.

Liberatore [10] proposed another interesting approach using GE and Flocking strategies to develop a Swarm-type intelligent for the Pac-Man ghost's controllers.

3 Ms. Pac-Man vs. Ghosts

Ms. Pac-Man vs. Ghosts (Figure 1) is a very popular arcade video game. In the version that we use [18], there is a set of four pre-generated toroidal 2D labyrinths, in which the ghosts and Pac-Man move. The ghosts always start in the "Lair", a rectangle in the middle of the map in which Pac-Man cannot enter, while Pac-Man starts in the bottom of the map.

Each labyrinth is composed of corridors and junctions filled with a lot of pills and four power pills. Both give points to Pac-Man as he walks over them, and

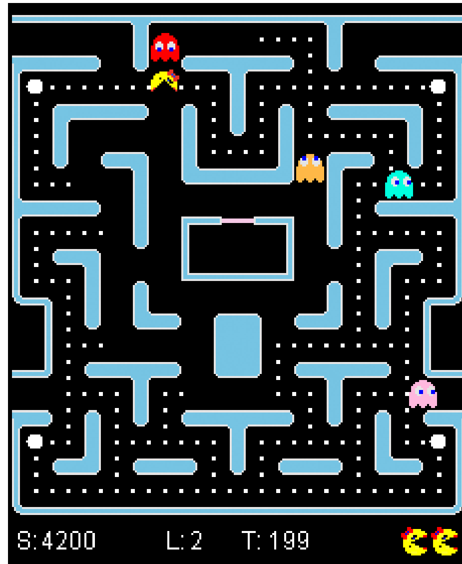

Fig. 1: Ms. Pac-Man vs. Ghosts videogame.

power pills make him able to eat the ghosts for a short period of time, while also slowing them. Eating ghosts will give Pac-Man points, earning some extra ones if he eats various ghosts in a row during the same power pill buff duration.

Pac-Man will try to eat all the pills and power pills to advance levels, while avoiding the ghosts, which will try to hunt him, making him lose a live when they walk over him. A level is completed when there are no pills and power pills left, and there are an infinite number of levels, repeating the same set of 4 labyrinths consecutively.

Pac-Man will strive to eat all the pills and power pills in the map to advance levels, while trying to get as many points as possible (eating any ghosts he can), since each 10.000 points achieved he gets an extra life. The game ends either when Pac-Man loses his three lives or after 24.000 turns, considering a turn passes every time both Pac-Man and the ghosts make a movement.

Ms. Pac-Man vs. Ghosts has been used in different competitions in which participants have used several different AI techniques in order to create the best automatic player. The most important competitions are *cha. Pac-Man Competition* [11] and *Ms. Pac-Man Vs. Ghost Team Competition* [1] [14]

Both the ghosts and Pac-Man use controllers to determine which movement is the best to make every turn. These controllers are the ones that must be implemented to participate in the competitions, which can be done using any technique available.

Every turn the game provides the controllers with its current state, so that the controller can seek relevant information it needs to choose a movement. Such information includes the pre-calculated distance from a point of the map

```
if (dist_closest_NE_ghost < 10) { escape }
else { seekFood }
```

Fig. 2: Example of a simple controller based on a decision tree.

to another (useful to check distances between Pac-Man and the ghosts), which is the movement that puts you further away from any other position (useful to run away from the ghosts), which movements are possible given a position, testing if a position is a junction or not, etc.

The original code of Ms.Pac-man vs. Ghosts also provides various examples of controllers like random movements for Pac-Man and the ghosts, aggressive behavioural ghosts, or a basic one for Pac-Man that considers whether the surrounding ghosts are edible or not. In particular, we use two different ghost's controllers in our experiments:

- *Random*: ghosts make random decisions.
- *Legacy*: it tries to reproduce the behaviour in the original game where each ghost used different heuristics to move.

4 A bot based on grammatical evolution

We use Grammatical Evolution (GE) to evolve Pac-Man controllers which play the game automatically. The types of programs that we consider valid Pac-Man controllers are defined using a context-free grammar. This way, we can reduce and refine the search space the evolutionary algorithm will explore, and focus on a certain type of programs that we think more promising. In particular, we are going to evolve decision trees in which the internal nodes check game conditions, and the leaves describe specific actions to execute in the game.

For example, the program in Figure 2 describes a very simple bot that run when there is a ghost near, and moves towards the closest pill in other cases.

In the general case, conditional statements can have other conditional statements inside and, therefore, the bot can take decisions based on more complex analysis of the game state. Note that, although decisions trees only allow to define reactive bots (we cannot encode explicitly advanced strategies consisting of action sequences), the evaluation of these decision's trees at every game turn can produce very complex behaviours.

In order to implement a Pac-Man bot based on GE, we use the JECO framework [17] (*Java Evolutionary COmputation library*) which supports different evolutionary computation techniques, including simple and multi-objective grammatical evolution.

4.1 Grammar design

We use two different types of terminals in our grammars:

```
<grammar>  ::= <sel-stat>
<sel-stat> ::= if(_<cond>_){_<stat>_}_else{_<stat>_}
               | if(_<cond>_){_<stat>_}
<stat>     ::= <action> | <sel-stat>
<action>   ::= escape | attack | seekFood
<cond>     ::= <num-st>_<num-op>_<num>
<num-st>   ::= dist_closest_NE_ghost
              | dist_closest_E_ghost
<num-op>   ::= EQ | NE | LT | GT | LE | GE
<numb>     ::= 0 | 5 | 10 | ... | 40
```

Fig. 3: High-level grammar.

- **Actions**. They represent Java methods that return a concrete move for the bot to execute. This way, we can use both abstract behaviours like *escape* or concrete moves like *left*.
- **State providers**. They represent Java methods that return information of the current game state either as boolean or numeric values. For example, *dist_closest_NE_ghost* returns the distance to the closest non-edible ghost.

We decided to design two grammars. The first one includes high level actions (strategies coded in Java) and state providers, while the second one includes medium-level actions and state providers. Both grammars contain conditional statements (if / if-else), numeric constants and numeric operators (`==`, `!=`, `>`, `>=`, `<`, `<=`). Our goal is to work at different levels of abstraction and study the effect in both the search space and the optimality of the resulting program.

Figure 3 shows the high-level grammar that contains 3 high-level actions: *escape* (run from the closest ghost), *attack* (go to the closest ghost) and *seekFood* (go to the closest pill). It only contains 2 state providers: distances to the closest edible and non-edible ghost.

Figure 4 shows the medium-level grammar that contains 4 actions to run to the closest pill, power pill, edible ghost and run away from the closest non-edible ghost. This grammar has several more state providers in order to create more complex conditions.

Medium-level state providers: *Distance to the closest non-edible ghost, distance to the closest edible ghost, number of active power pills, distance to the closest pill, distance to the closest power pill, geometric mean of the distances to all non-edible ghost, geometric mean of the distances to all edible ghost*

Medium-level actions: *Run to the closest pill, run to the closest power pill, run away from the closest non-edible ghost, run to the closest edible ghost*

High-level actions: *escape, attack, seekFood*

- *escape*: Pac-Man moves towards the closest pill if he can reach it before any ghost, or runs away from the closest ghost if he can't (Also runs away if there are no power pills left).

```
<gram>      ::= <sel-stat>
<sel-stat>  ::= if(_<cond>_){_<stat>_}_else{_<stat>_}
              | if(_<cond>_){_<stat>_}
<stat>      ::= <action> | <sel-stat>
<action>    ::= run_to_closest_pill
              | run_to_closest_ppill
              | run_to_closest_E_ghost
              | run_from_closest_NE_ghost
<cond>      ::= <bool-st>
              | <num-st>_<num-op>_<num>
<bool-st>   ::= <bool-api> | not _<bool-api>
<bool-api>  ::= is_junction
<num-st>    ::= dist_closest_NE_ghost
              | dist_closest_NE_ghost
              | dist_closest_pill
              | ...
<num-op>    ::= EQ | NE | LT | GT | LE | GE
<numb>      ::= 0 | 5 | 10 | ... | 40
```

Fig. 4: Medium-level grammar.

– *attack*: Pac-Man moves towards the closest edible ghost if no other ghost can reach the edible one before him. If there are no edible ghosts, Pac-Man moves in the same direction as he did in the previous turn.

– *seekFood*: Pac-Man moves towards the closest pill if he can reach it before any ghost does. If there are no more pills, moves towards the closest power pill.

4.2 Operators and Fitness function

After several tests comparing selection, elite, crossover and mutation operators as well as their hyper-parameters, we obtained the best results using: Binary Tournament selection[4] with 5% elite, LHS crossover [7] with 60% probability and Integer Flip mutation with 10% probability and using Neutral mutation [12].

To maximize the score of the controller, we minimize the following fitness function (it will be reviewed in later experiments in section 5.1):

$$f = 100000 - score$$

Pac-Man obtains points every time he eats a pill (10 points), a power pill (50 points) or an edible ghost (200 points). When Pac-Man eats more than one ghost in a row, he gets extra points (400 for the second ghost, 800 for the third, ...). The fitness function will be reviewed in later experiments in section 5.1.

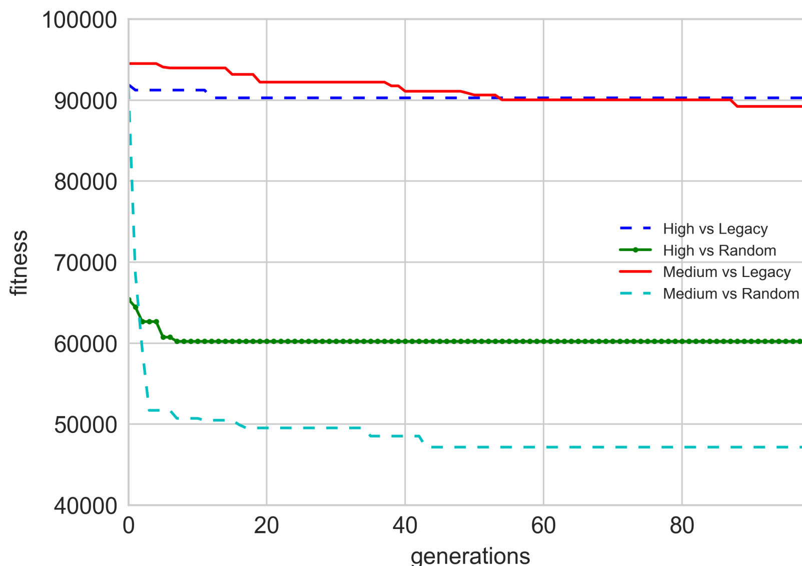

Fig. 5: Fitness evolution vs. generation (less is better) with different controllers.

4.3 Results

We performed four experiments evolving bots with the previous two grammars and using two different ghost controllers: Random and legacy. Note that the legacy ghosts are much more challenging than the random ghosts. We also compared our bots with the UCD Dublin bot [6], another bot trained using Grammatical Evolution and high-level actions and state providers.

All the experiments were run using the same configuration: Population size 100, Generations 100, 30 games played per individual evaluation, Binary Tournament selection, LSH crossover (60%), Integer Flip mutation (10%), Neutral mutation and elitism (5%).

Figure 5 shows the evolution of the fitness function as we produce new generations of individuals in each experiment. Table 1 displays the results of the experiments. In each experiment, we measure the final score, the level reached and the time played showing the maximum, average and standard deviation values of 1000 games. As we were expecting, the *Legacy* ghosts are more challenging opponents than the *Random* ghosts, and all the values are smaller because the games are much shorter.

Both our grammars can produce bots that play better than the baseline controllers (a random controller and other one that always go towards the closest pill). Besides, our bots play better than the UCD Dublin bot, and that is interesting because all of them are created using Grammatical Evolution. This seems to happen due to the usage of excessively specific functions, which limit its behaviour, i.e. forcing Pac-Man to wait next to a power pill. Their bot also uses

Table 1: Pac-Man vs Ghost controllers' comparison.

Pac-Man	Ghosts	score			level			time (game ticks)		
		max	avg	std	max	avg	std	max	avg	std
Random	Random	1380	501	213	1	0.036	0.186	5635	1943	887.5
NearestPill		18910	4471	2654	5	1	0.9	7216	1795	1018
UCD Dublin bot[6]		11640	4288	-	-	-	-	-	-	-
Medium-level		**64600**	**48558**	**10780**	**18**	**15**	**3.4**	**24000**	**21579**	**4470**
High-level		**55480**	**32704**	**13237**	**18**	**10.4**	**4.3**	**24000**	**17457**	**6784**
Random	Legacy	1840	197	107	0	0	0	877	465	61.3
NearestPill		7190	3531	638	1	0.4	0.5	1881	1152	143.7
UCD Dublin bot[6]		12350	3945	-	-	-	-	-	-	-
Medium-level		**15960**	**6358**	**2883**	**3**	**0.9**	**0.7**	**4973**	**1916**	**730**
High-level		**20040**	**5972**	**2832**	**4**	**1**	**0.6**	**8364**	**2026**	**1020**

large amount of parameters like dimensions of a frame (centered on Pac-Man) to evaluate certain conditions. Conversely, we make use of path distances and numeric operators, resulting in a less complex game status analysis.

Using the medium-level grammar, we obtain better results than using the high-level grammar in average, probably because the actions and state providers make possible to create controllers that exploit scenarios that cannot be exploited using the high-level grammar. However, the high-level grammar obtains better results in some particularly good games (max. values). The best evolved controller using the medium-level grammar playing against the Legacy ghosts is able to obtain 6358 points and complete almost 1 level in average. In the best games, this same controller is able to obtain 15960 points and complete 3 levels.

When we analysed the behaviour of the evolved controllers, we discovered that the bots generated by the high-level grammar share almost always the same code, and achieve slightly lower scores, eating pills conservatively by avoiding ghosts. However, the medium-level grammar tends to generate bots which manage to get stuck next to power pills (stopping themselves), wait for the ghosts to be close and proceeding to eat first the power pill and then the ghosts. This hunter behaviour allows Pac-Man bots to achieve notable scores, because there is a multiplicative bonus when Pac-Man eats several ghosts in a row.

We also made experiments playing against the *Starter* ghosts controller, an implementation that run away when the ghosts are edible and chase Pac-Man with certain probability when they are not edible. Most of the executions using the medium-level grammar evolved controllers able to exploit a bug in the code. They went in circles forever in a corner of the board completing levels (a level can be completed just waiting for enough turns) and not being chased by the ghosts.

Regarding the type of generated programs, Figure 6 shows an example of the controller evolved using the medium-level grammar against the random ghosts. Basically, Pac-Man runs away from non-edible ghosts when they are very close, goes towards the closest power pill when the ghosts are close (but not very close), and eat pills in other cases.

```
if (dist_closest_NE_ghost > 10) {
    if (dist_closest_NE_ghost < 20) {
        run_to_closest_ppill
    } else { run_to_closest_pill }
} else { run_from_closest_NE_ghost }
```

Fig. 6: Controller evolved using the medium-level grammar with random ghosts.

5 Multi-objective optimization

Multi-objective optimization arises when a single objective may not adequately represent the problem being faced, so modelling it with several objectives is preferred. In multi-objective optimization, there are therefore n different objectives each one with its own fitness function f_i:

$$f_i(X)|i = 1, ..., n$$

The difficulty of this kind of problems lies on determining which individual optimizes all the objectives better. We say that a solution is a Pareto optimal if none of the objective functions can be improved in value without degrading some of the other objective values [3].

Exist a wide variety of methods when implementing a multi-objective algorithm. The easiest consists in creating a fitness function which is a linear combination of all other functions to optimize. The main problem concerns the difficulty to find the adequate weights.

Another popular approach that we will use in this work is the *NSGA-II* [8] algorithm, which delivers very good results yet is computationally expensive, especially for large populations.

5.1 Why to apply it

Using a single objective function that maximizes the score, inevitably leads to bots that keep moving around a power pill until one or more ghosts approach, at which point Pac-Man eats the power pill and proceeds to eat as many ghost as possible, exploiting the ghost score multiplier.

This strategy only works when there are still power pills available. With no power pills left Pac-Man keeps rambling until a ghost eats it, and the game is over. The consequence is that the controller usually is not able to complete more than one level. In order to complete more, we tried to model a multi-objective problem with two different fitness functions:

$$f_1 = 100000 - \text{score}$$
$$f_2 = 100 - \text{last level reached}$$

With this functions, we create a set $F = [f_1, f_2]$ that will be used by NSGA-II to evolve the population.

Table 2: Pac-Man vs Ghost controllers' comparison including Multi-Objective.

Pac-Man	Ghosts	score			level			time (game steps)		
		max	avg	std	max	avg	std	max	avg	std
Medium-level 1	Random	64600	48558	10780	18	15	3.4	24000	21579	4470
Medium-level (MO)		**62050**	**46922**	**1243**	**18**	**15**	**4**	**24000**	**20868**	**5094.5**
High-level 1		55480	32704	13237	18	10.4	4.3	24000	17457	6784
High-level (MO)		**57370**	**32441**	**12712**	**17**	**10**	**4.1**	**24000**	**17536**	**6604.7**
Medium-level 1	Legacy	15960	6358	2883	3	0.9	0.7	4973	1916	730
Medium-level (MO)		**18020**	**6229**	**2832**	**3**	**0.9**	**0.7**	**5041**	**1905**	**725**
High-level 1		20040	5972	2832	4	1	0.6	8364	2026	1020
High-level (MO)		**20040**	**5972**	**2832**	**4**	**1**	**0.6**	**8364**	**2026**	**1020**

5.2 Results

Table 2 shows the results with and without multi-objective. 1 refers to *f1* and MO to *F=[f1,f2]*. Unfortunately, multi-objective optimization does not seem to obtain better results in our problem in terms of the number of completed levels. Since both our objectives depend directly or indirectly on the score multi-objective evolving doesn't produce a diversity of behaviours in our programs, making us believe that multi-objective optimization works at its best in situations where goals are not directly related.

Our results also show that even if we force the objective of reaching more levels, the controllers obtain similar scores since Pac-Man advances levels by eating all pills in the board (hence getting high scores). The same happens in the opposite way: if we focus on points, Pac-Man will complete as many levels as possible, because it aims to eat all pills.

6 Conclusions and future work

In this article, we have presented a grammatical evolution based evolutionary algorithm to generate Pac-Man bots using some different levels grammars. The bots produced are capable of acquiring high scores and completing several levels. Those results are better than the included hand-coded bots as well as other known evolutionary bots[6].

After investigating the effects of multi-objective optimization on Pac-Man, we can conclude that it is not very useful in this context. Since every sub-goal we can think of is dependant on the score, multi-objective evolving doesn't produce a diversity of behaviours in our programs, making us believe that works at its best in situations where goals are not directly related.

Nevertheless, there is always the advantage of guiding the search process with multi-objective, the same way we do with a well-designed grammar. It can provide the bot desirable supplementary behaviour through the pursue of side objectives. For example, staying as far as possible from the ghost, which grants higher survivability.

We used JECO (*Java Evolutionary Computation Library*) as a base framework. All the code base is located within a git repository[1]. Future work is focused on adding more advanced Artificial Intelligence techniques to the current comparison, namely Behaviour Trees, NEAT algorithm or Grammatical Swarm, maybe with some kind of hybridization.

References

1. Ms. Pac-Man Vs. Ghosts Tournament, http://www.pacmanvghosts.co.uk/
2. Alhejali, A.M., Lucas, S.M.: Evolving diverse Ms. Pac-Man playing agents using Genetic Programming. In: Computational Intelligence (UKCI), UK Workshop on. pp. 1–6. IEEE (2010)
3. Araujo, L., Carlos, C.: Algoritmos evolutivos: Un enfoque práctico. RA-MA (2009)
4. Blickle, T., Thiele, L.: A mathematical analysis of Tournament Selection, pp. 9–16. Morgan Kaufmann (1995)
5. Brandstetter, M.F., Ahmadi, S.: Reactive control of Ms. Pac Man using information retrieval based on Genetic Programming. In: Computational Intelligence and Games (CIG), IEEE Conference on. pp. 250–256 (2012)
6. Galván-López, E., Swafford, J.M., O'Neill, M., Brabazon, A.: Evolving a Ms. Pac-Man controller Using Grammatical Evolution, pp. 161–170. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
7. Harper, R., Blair, A.: A Structure Preserving Crossover in Grammatical Evolution. IEEE Congress on Evolutionary Computation (Sep 2005)
8. Kalyanmoy Deb, Associate Member, I.A.P.S.A.T.M.: A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation, Vol. 6, No. 2, April (2002)
9. Koza, J.R.: Genetic Programming: On the programming of computers by means of Natural Selection, vol. 1. MIT press (1992)
10. Liberatore, F., Mora, A.M., Castillo, P.A., Guervós, J.J.M.: Evolving Evil: Optimizing Flocking Strategies Through Genetic Algorithms for the Ghost Team in the Game of Ms. Pac-Man, pp. 313–324. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
11. Lucas, S.M.: Ms. Pac-Man Competition (2007-2011). http://dces.essex.ac.uk/staff/sml/pacman/PacManContest.html
12. Oesch, C., Maringer, D.: A Neutral Mutation Operator in Grammatical Evolution. Advances in Intelligent Systems and Computing Intelligent Systems p. 439–449 (Sep 2014)
13. O'Neill, M., Ryan, C.: Grammatical Evolution Evolutionary Automatic Programming in an Arbitrary Language. Springer-Verlag New York Inc (2012)
14. Piers R. Williams, Diego Perez-Liebana, S.M.L.: Ms. Pac-Man Versus Ghost Team CIG Competition. IEEE Computational Intelligence and Games (2016)
15. Poli, R.: A Simple but Theoretically-Motivated Method to Control Bloat in Genetic Programming, pp. 204–217. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
16. Poli, R., Langdon, W.B., McPhee, N.F., Koza, J.R.: A field guide to Genetic Programming. Lulu Press (2008)
17. Risco, J.L.: JECO Library, https://github.com/jlrisco/jeco
18. Robles, D.: Pacman vs ghosts simulator, https://github.com/davidrobles/pacman-vs-ghosts

[1] https://github.com/hecoding/Pac-Man