Coevolutionary Strategies for Super Mario Bros.

Jan van de Kerkhof, Jan Paredis
Department of Knowledge Engineering,
University of Maastricht
P.O. Box 616, 6200 MD Maastricht
The Netherlands

Abstract—**This paper introduces a coevolutionary genetic algorithm for the development of neural network based controllers for the game Infinite Mario Bros. Using a dual population structure, in which (partial) tests are evolved alongside solutions (e.g.agents playing playing the game). An agent and test representation is proposed to optimize the algorithm. Incremental Test Development (ITD) is proposed to further evolve the test population. ITD gradually increases the difficulty of tests during tyhe coevolution. The agents evolved by the algorithm perform significantly better than controllers evolved by traditional single population genetic algorithms. A lack of increase in performance after a test difficulty of 3 explains why agents evolved with ITD do not outperform the agents evolved without it. Nonetheless ITD might be useful for other games, where learning extends over more levels of difficulty.**

I. Introduction

While apt research has been done into the application of genetic algorithms for board games like Backgammon [8], Go [3] and Checkers [1], research into the subject of video games is now on the rise. Real-time video games provide an ideal test bed for genetic algorithms. Video games provide a "smooth and long learning curve" [12]. Only by practice and repetition can a human player become a master in the game. Video games require the player to apply different sets of skills at different times, usually in rapid succession. It is this last property of video games that makes neural networks an attractive representation for agents, since neural networks provide a mapping from complex input spaces to complex output spaces [9].

Coevolutionary algorithms are optimalisation methods inspired by predator-prey relations in nature. They have been used to evolve controllers which apply effective strategies in the context of real-time strategy games like Supreme Commander [4] and the flight game XPilot [7].

In this paper a modified version of the game *Infinite Mario Bros.*, developed by Markus Persson [10], is used as a test bed. *Infinite Mario Bros.* is an all java adaptation to the popular platform game *Super Mario Bros.*, released by game developer Nintendo in 1985 [13]. *Super Mario Bros.* is a real-time, two-dimensional platform game. The game revolves around the character Mario, who has to get to the end of a *level*. These *levels* have a varying *difficulty*. The goal is for Mario to get to the far end of the level, starting at the left-most point and ending at the right-most point. Levels consist of surfaces of different kinds and sizes (platforms) and contain gaps and enemies. The higher the *difficulty*, the more gaps and enemies a level will contain. When Mario falls into a gap he dies, and when Mario is hit by an enemy three times he also dies. The agent (Mario) can perform several actions. At each time step the agent can choose between moving left, right and down. He can also jump and speed. If Mario is in the Fire state, speeding results in throwing a fireball that can kill enemies. The movements are mutually exclusive, yet jumping and speeding can be combined with all movements.
What makes *Infinite Mario Bros.* an interesting game for the application of genetic algorithms, is the high dimensionality of the input space. The input space differs highly at different stages in the game. Mario has to cope with a combination of platforms, gaps and enemies at the same time. The output space, be it less complicated, still provides a lot of different possibilities. The combination of movements (left, right and down) and actions (jumping, speeding, no action, and no move) provides 4 x 4 = 16 different outputs.
Togelius et al. [12] have set a benchmark for the application of evolutionairy algorithms in the game by developing several controllers with different network layout, by means of different genetic algorithms. They have made their adapted framework available for further research into the subject. They have also hosted an AI competition for the game four years in a row [2], [11].
Paredis [5] introduces an algorithm for the coevolution of feed-forward neural networks. This algorithm is used to train networks in the context of two-dimensional classification problems, amongst others. Paredis' research shows that the algorithm is significantly faster in reaching proper solutions than regular, single population, genetic algorithms. The algorithm reaches solutions not reached by traditional single population evolutionary algorithms [6]. The current paper introduces an adaptation to the algorithm in [5], [6] by providing a partial test representation and an agent architecture for *Infinite Mario Bros.*. Incremental Test Development, where members of the test population with a low fitness are incrementally replaced by tests with a higher difficulty, is proposed as a way to fully evolve the test population and improve the algorithms' performance.
The structure of this paper is as follows. Section 2 explains the game environment. Section 3 covers the implementation and introduces the algorithm, the agent architecture and the test architecture. At the end of section 3 Incremental Test Development is introduced. Section 4 contains the test set-up and test results. In section 5 ideas for future research are

given. Finally, section 6 draws conclusions from these results.

II. Game Environment

In *Infinite Mario Bros.*, the main goal is for Mario to reach the end of a level. This can be hindered by Mario falling into a gap or getting killed by enemies. Auxiliary goals are collecting coins that are spread out across the level; killing as many enemies as possible or completing the level in the least amount of time. There are several different difficulties the game can be played on. The difficulty of a level (the stretch of ground Mario has to clear) determines the amount of gaps and enemies that level contains.

There are several different modes Mario can be in. At the start of each level, Mario is in the Fire mode. During the course of the level, when Mario is hit by an enemy, he goes into the Big mode. If he is hit again, he goes into the Small mode. This last mode is the most vulnerable mode, since Mario is one hit away from death. Big implies that Marios height spans two squares, opposed to a single square when Mario is in the Small mode. In the Fire mode, Mario is able to shoot a fireball when pressing the speed button. These fireballs kill enemies on impact, but only two can be active at the same time. Regardless of which mode Mario is in, when he falls into a gap, he dies. There are several different types of platform Mario can encounter. Apart from the ground Mario walks on, there are floating platforms Mario can pass through from the bottom and coin blocks which Mario can destroy (and collect a coin) by jumping up into them.

Finally, there are the enemies that Mario has to either evade or kill. There are three types of 'stompable' enemies, enemies that can be killed by jumping on them. The first one is the *Goomba*, a small, round enemy. The second one is the *Koopa*, a slightly larger, turtle-like enemy. When a Koopa is stomped on, he pulls back into his shield. Mario can pick this shield up and use it to kill other enemies. Thirdly, there is the *Bullet Bill*, a horizontally flying bullet, which is shot from cannons that block Marios' path. The two enemies that cannot be killed by stomp are the *Spiky*, a walking shield with spikes on it, and the *Piranha Flower*, a biting flower that jumps up from flower pots placed across the level. The Piranha Flower can be killed by jumping up against its bottom.

The Goomba, Koopa and Spiky type also have a winged version. These can fly around and lose their wings when hit, reducing them to a regular, walking, enemy. Except for the Spiky, throwing a fireball (by pressing speed in the Fire mode) will kill an enemy.

III. Implementation

The algorithm used in this paper is a coevolutionary genetic algorithm (CGA) introduced by Paredis [5]. It has been proven effective in classification and constraint satisfaction problems, amongst others.

The algorithm simulates evolution based on a 'predator-prey' model. This implies that two populations have an inverse fitness relation when they compete with each other for survival. Succes for one species means failure for the other species. Therefore, adaptation of one population results in lesser performance of the other, which in turn forces the other population to adapt. This constant interaction between populations leads to an incremental, accelerated search for fitting solutions and reaches solutions regular, single population, genetic algorithms (GAs) aren't able to find.

In contrast to most GAs, which use a single solution population, Paredis' algorithm evolves two populations simultaneously. The *solution* population, consisting of the networks being evolved, and the *test* population, consisting of the set of problems the networks are being evolved and tested on. Both populations are kept sorted on fitness for reasons which will be explained in the next section.

Another thing that stands out from traditional GAs is the fitness calculation. Instead of calculating a single fitness value based on the same, predefined tests, Paredis introduces *lifetime fitness evaluation* (LTFE). He argues that fitness evaluation in nature is "far more partial but continuous" [6], stating that an individuals' fitness is determined by multiple *encounters* during the span of its lifetime. These encounters arise from a complex test environment and are not predetermined. With LTFE, an individuals' fitness is averaged over the last twenty of these partial encounters. Since the populations have an inverse fitness relation, the payoff for a test is calculated by taking the inverse of the payoff the agent has received from an encounter. The fitness calculation will be explained in detail in section III.D.

A. The Coevolutionary Algorithm

The main cycle of the algorithm is based on *traditional*, single population GAs. At each iteration two parents are selected from the solution population. This selection is biased towards individuals with a higher fitness. This is possible because both populations are sorted on fitness. In this paper, this bias is set linearly, such that the individual with the highest fitness has twice the chance of being selected of the individual with the lowest fitness.

From the two selected parents, a child is created. By means of mutate-crossover, the genetic code of the parents is combined to create a child. In this paper, 12-point recombination is used as the crossover method, which proved to reach proper solutions the fastest in [6]. As mutation operator, adaptive mutation is used. Adaptive mutation implies that a weight in a network is mutated dependent on the value of that weight in both parent networks. If the difference between two weights in both parents is larger than 0.01, that weight has a 0.001 chance of mutation. If the difference is smaller than 0.01, the mutation chance increases linearly to 0.02 as the difference gets closer to 0. If a weight is mutated, it is replaced by a random number from the interval [-100,100].

After a child is created, it is subjected to testing and its fitness value is calculated. If the child is fitter than the lowest member of the solution population, this lowest member is removed from the population and the child is inserted into the correct rank.

The coevolutionary algorithm extends this basic structure by introducing the lifetime fitness evaluation and a dual population structure. With LTFE, each solutions' fitness value is now defined by the *encounters* it has had in its lifetime. Each of the solutions has a history of the encounters it has had. The fitness value is calculated by taking the average payoff over the last twenty encounters. The same holds for each test in the test population.

The entire *coevolutionary* algorithm can now be explained. First, the two populations are initialized. For the solution population, this is done randomly. For each weight of each network in the solution population, a random value from the interval [-1,1] is chosen. The tests are part of the problem specification and are known beforehand, so the test population does not have to be randomized. The fitness of each initial solution is calculated by testing it against twenty randomly selected members of the test population. The initial fitness of each test is calculated in the same way.
After initialization, the main cycle is started (see the pseudocode given below). At each iteration of the cycle, twenty encounters between a solution and a test are simulated. The resulting evaluation is pushed into the solutions' and the tests' history. Note that the function toggle implements the inverse fitness relation. The respective fitness values are then updated. Since each population is sorted on fitness, the position of each test and solution in its population is updated as well. After these twenty encounters, the algorithm creates a (solution) child. In case this child is fit enough, it is inserted in the solution population at its apprpriate rank.

```
for 20 encounters do
    Sol = SELECT(sol-pop);
    Test = SELECT(test-pop);
    Res = ENCOUNTER(sol, test);
    UPDATE-HISTORY-AND-FITNESS(sol, Res);
    UPDATE-HISTORY-AND-FITNESS(test,
    toggle(Res));
end
Sol1 = SELECT(sol-pop);
Sol2 = SELECT(sol-pop);
Child = MUTATE-CROSSOVER(sol1, sol2);
F = FITNESS(child);
INSERT(Child, F, sol-pop);
```

B. Agent Representation

Togelius et al. [12] evolve several kinds of networks, but use the same input for each network. The input of the networks are two binary inputs indicate whether Mario can jump and whether Mario is on the ground plus a number of sensors for neighbouring squares. There are two sets of sensors, one for enemies and one for environmental obstacles.

They made a distinction between a small network, which uses an inner ring of 9 sensors; a medium network, which uses an inner and middle ring of sensors and a large network, which uses all 49 sensors as input. Togelius et al. [12] show that the effectiveness of the GAs drops as the networks get bigger. The networks with the least amount of input from the sensors evolved best. A problem they faced, however, was the short-sightedness of the agents. The agents often did not see an enemy coming or jumped into a gap because their sensors did not perceive it. Another problem they faced, because of the discrete input, was that the agents were not able to determine their position exactly and had trouble timing jumps and avoiding enemies in tight spaces.

Considering the high level of correlation between the blocks in a level of Infinite Mario Bros, a possibly better, compressed, representation of the game state is proposed here. As smaller networks produce better solutions, the game environment needs to be represented by the least amount of values.
Each level consists of four different types of elements the agent needs to take into account. These are: platforms, enemies, obstacles and gaps. Each of these elements can be represented by at most four floating point numbers, largely decreasing the amount of values needed to represent the entire game state.

For each platform, the starting x-coordinate, the starting y-coordinate, the length and the type are necessary. For enemies, this is the x-coordinate, the y-coordinate and the type of enemy. A gap consists of the starting x-coordinate, starting y-coordinate, ending x-coordinate and ending y-coordinate. For an obstacle (flower pot, wall or cannon), it does not matter which type it is, since the agent cannot pass through either way. The only input values necessary for an obstacle are the upper y-coordinate and the x-coordinate. The input space can be limited even more. When Mario is currently on a platform, it does not matter which type it is, or where it started. The only data necessary is the remaining length of the platform. When Mario is underneath coin blocks, the same holds, since the agent only needs to know if, and when, he can fully jump. Hence, there are four values which are necessary as input for every network, namely

1) The mode Mario is in (2, 1 or 0)
2) Is Mario carrying (1 or 0)
3) Can Mario jump (1 or 0)
4) Is Mario on the ground (1 or 0)

The last two of these inputs were also given to the networks in [12]. Using standard *feed-forward neural networks* [9] (containing a bias node and using the sigmoid function) as a representation of the agent, the total number of input nodes will then be :

$$1 + 4 + 2 + 4x_1 + 3x_2 + 4x_3 + 2x_4$$

where 1 + 4 + 2 stands for the bias node, the four required values and the two platform lengths and x_1, x_2, x_3 and x_4 stand for the amount of platforms, enemies, gaps and obstacles, respectively, the agent is allowed to perceive.

With this representation, a network can be scaled by allowing the agent to perceive more elements of a certain type. To retain association between two different game states, the respective input values need to always be entered at the same nodes in the networks input space. This can be done by ordering each value according to the distance to the agent. Every coordinate is shifted in order to represent the (floating point) distance to the agent. For each element, it also matters whether it is in front or behind the agent, since elements the agent has passed usually do not matter for the continuation of the level.

C. Test Representation

The algorithm uses a *partial but continuous* fitness evaluation. Therefore, the problem space needs to be transformed into a more *partial* representation, for the algorithm to have its desired effect. In each level, there will be different parts which an agent will have more difficulty completing. By splitting a level into multiple *segments*, i.e. levels which are significantly shorter, these different bottlenecks can be isolated. This way, when the agents in the population have learned to easily traverse segments with certain properties, the ones they haven't yet mastered will be the ones they will be trained on the most.

At initialization, partial tests can be generated by creating several levels of large length. These long levels are then cut segment by segment, at each time creating a partial test. These segments need to satisfy two criteria.

The first criterium is that the segment cannot end or start in a gap. Doing so will either make the segment impossible to finish or will train the agent to believe that it is good to jump into gaps. Also, the agent needs to have a few "blocks" of distance to the first gap in order to gain enough speed to jump it.

The second criterium is that the agent cannot start in, or very close to, an enemy. This will result in Mario getting hit right off the block, which gives a test an unfair advantages over others.

In order to ensure that these criteria are met, each segment will have a safe span of five blocks at the start, where no enemy or gap will be present.

D. Fitness Calculation

In order to calculate the fitness of an agent and the inverse fitness of a test there are several different possibilities. The calculation measure that seems most fitting, which is also used in [12], is the *distance travelled* by the agent, assuming that the further the agent traverses a level, the better that agent is. However, with partial testing, this calculation measure is a lot less effective. When a level is split into different segments, these segments are considerably shorter than a full level.

With smaller segments, the chance that an agent dies decreases as the segment shortens, at the same time reducing the ability of distance travelled to discriminate between agents.

To be able to fully calculate an agents payoff (the higher the better), the calculation measure needs to distinguish between the following cases :

1) The agent commits suicide by jumping in a gap
2) The agent is killed by multiple hits from enemies
3) The agent gets stuck and time runs out
4) The agent finishes the segment with one or two hits
5) The agent finishes the segment flawlessly

From a survivalist perspective, committing suicide is the ultimate sin. Straight after that comes getting stuck. Getting hit by an enemy is acceptable as long as it does not happen three times in a row. Still taking into account the distance the agent has travelled the following heuristic emerges. If an agent commits suicide, he receives 0 payoff. When he finishes a level, he receives 0 payoff when in the Small mode, 1 when in the Big mode and 2 when in the Fire mode. Since the agent still has to be rewarded for making progress, if he did not commit suicide, he also receives the distance travelled divided by 1333. This number is motivated by the fact that levels are usually 4000 units long and Mario is able to get hit 2 times during the course of a level. This way, clearing a third of a level compensates for getting hit once. The (inverse) payoff for a test can be calculated by taking the maximal value the agent can possible receive on the segment and subtracting it with the payoff the agent has received.

E. Incremental Test Development

In the algorithm presented in III.A, the test population does not truly evolve. Tests with higher fitness get more encounters, but the members in the test population never change, only their ranking changes. *Incremental Test Development* (ITD), as presented in this paper, tries to improve upon the test space as the agent population evolves.

During the course of a run of the CGA, the fitness value of the tests drops as the agents start mastering the segments. ITD sets a minimum on the fitness value a test is required to have. When a test is due for replacement, it is replaced by a segment of one difficulty higher.

After each iteration of the algorithm (the creation of a new child), ITD checks which of the tests fail to achieve this minimal fitness and then attempts to replace each of these segments by a segment of one difficulty higher. These new tests are evaluated in the same way as the other tests and only added to the test population if they satisfy the minimal fitness requirements.

For the sake of noise reduction towards this minimal threshold, the distance travelled by the agent is removed from the heuristic. With ITD, the test either gets payoff 3 (agent dead or stuck), 2 (agent completed with one hit), 1 (agent completed with two hits) or 0 (agent completed flawlessly).

If the test population is initialized at difficulty 0 and the right minimal fitness value is chosen for the tests, ITD will slowly make the agents move up in level difficulty. It is expected that the algorithm with Incremental Test Development will provide better results than the algorithm without ITD.

IV. Testing

In order to properly compare the best agents obtained from test runs, the same testing is applied as in [12]. The best agents are tested against 1000 levels each of difficulty 0, 3, 5, and 10. From these evaluations, the average distance travelled by the agent is taken. A level of 250 blocks (each block consists of 16 pixles) stands for a distance of 4000 units/pixels. This

is the maximum distance an agent can get on a level. It is good to note that difficulty 0 does not contain gaps and only enemies of the Goomba and Koopa type. Difficulty 3 contains gaps and every type of enemy and difficulty 5 and 10 only differ in that they contain more gaps and more enemies.
In table I the test results from [12] are shown. This table shows the performance, i.e. average distance travelled, of the feed-forward neural networks evolved in the testing.
As can be seen from table I, the best neural network based

Controller	0	3	5	10
Small network	1784	719	606	531
Medium network	864	456	410	377
Large network	559	347	345	300

TABLE I: Test results from [12], showing the average distance the different agents travelled at difficulty 0, 3, 5 and 10

agent in [12], which was the small network based agent, on average reached less then half the maximum distance at difficulty 0 and around an eighth of the maximum distance at difficulty 10.

Table II contains the first batch of test results from the CGA. These results are obtained from runs of the CGA with ITD disabled. The solution population size is set to 300, and the test population size is set such that the total length of the segments being tested on is equal to a total distance of 80 levels of 4000 units (or 250 blocks). With a segment length of 50 blocks, this means that the test population size is set to 400. Since [12] showed that smaller networks evolve better, the agents are set to perceive 3 platforms, 2 enemies, 1 gap and 1 obstacle. No input from what is behind the agent is given. The number of nodes is: 31 input nodes, 12 hidden nodes and 5 output nodes; one for each possible action. The network size is comparable to that of the small network in table I, which has a total of 22 input nodes, 10 hidden nodes and 5 output nodes. In order to remove noise from the testing, the random level seeds are kept the same throughout each run. At each run, the CGA lasts for 50000 cycles. After every 500 cycles, one of the agents from the population is benchmarked against 10 levels each of difficulty 0, 3, 5 and 10. The agent that is benchmarked is selected by testing each of the agents in the population against 5 levels of difficulty 5 and selecting the agent that reached the furthest distance.
Table II shows the avarage distance travelled by the agent that tested best on the benchmark at the end of 50000 cycles. The results are averaged over 4 runs for every setting. The first column shows the difficulty the test population is initialized on and the second column shows the segment size in blocks. The best results from the CGA were obtained by initializing the test population at difficulty 5 and setting the segment size to 50. Table III shows the best agents obtained from the CGA against the best agents obtained in [12].

As can be seen from this comparison, the best agents obtained by runs of the CGA travel more than twice the distance of those obtained in [12].

The next set of experiments investigate the influence of the

Test Difficulty	Segment size	0	3	5	10
0	50	3136	825	725	584
0	100	3528	1046	945	747
3	50	2995	1290	1080	825
3	100	3410	1415	1185	935
5	50	3653	1828	1538	1174
5	100	3778	1594	1374	1081
10	50	3546	1637	1371	1086
10	100	3775	1744	1461	1155

TABLE II: Results from testing the CGA with different test difficulties and segment sizes

Agent	0	3	5	10
Small network [12]	1784	719	606	531
CGA at difficulty 5 and segment 50	3653	1828	1538	1174

TABLE III: Best network from [12] against the best agent from the CGA

difficulty of the tests on the performance. Figures 1 and 2 show the benchmark scores at difficulty 0 and 5 of runs initialized at segment length 50, where each plot represents a run where the test population is initialized at difficulty 0, 3, 5 or 10. These runs were chosen for comparison, as these single runs obtained the best results.

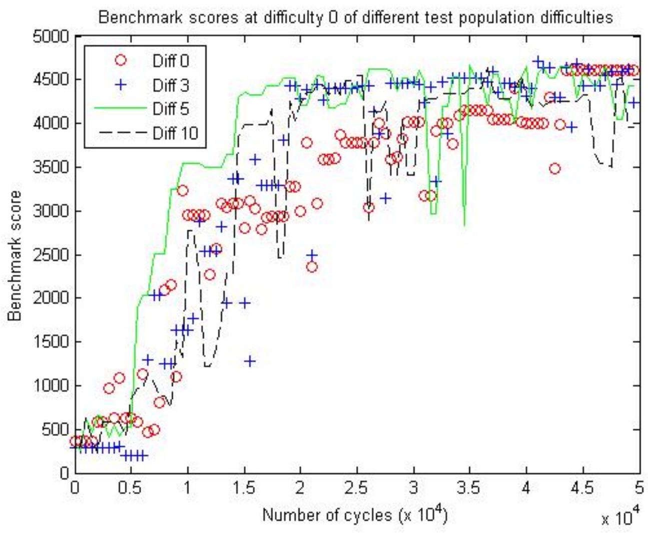

Fig. 1. Benchmark scores at difficulty 0 against the number of cycles

These figures show that for an agent to perform well at difficulty 0 it does not matter which difficulty the test population is initialized at. The run with the test population initiliazed at difficulty 5 have a slightly faster learning curve, but reaches the same optimum (which is the maximal value possible on the benchmark). Fig. 2 shows the benchmark scores at difficulty 5, it can be seen that the runs initialized at test difficulty 3, 5 and 10 behave in a similar manner,

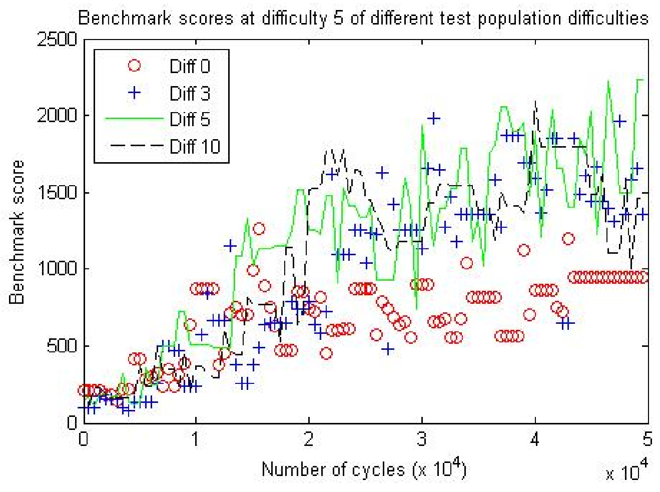

Fig. 2. Benchmark scores at difficulty 5 against the number of cycles

converging around a score of 1500, but the run initialized at test difficulty 0 fails to reach the same optimum, converging around a score of 1000.

Looking at the behaviour of some of the best runs at every test difficulty, it is clear that the CGA behaves similarly from test difficulty 3 and up. The only significant change in performance arises when the test population is initialized at difficulty 0. This decrease in performance can be explained by the fact that difficulty 0 does not contain any gaps and only a few of the enemy types encountered at higher difficulties. The agent therefore misses training on elements encountered at these higher difficulties. Elementary behaviour such as clearing a gap and avoiding an enemy can already be learned properly at difficulty 3. An increase in the amount of elements the agent encounters (by raising the test difficulty) does not make a difference in performance. At difficulties higher than 3 nothing new is to be learned. This can be explained by looking at the agent structure. Since the agent only perceives a few elements at a time, its behaviour is limited to what it perceives. While levels of higher difficulty contain more gaps and enemies, the agent only perceives the ones closest to it, making additional gaps and enemies redundant for training.

After these initial tests, the algorithm is run with ITD enabled. The segment size is kept at 50 blocks and the population size at 300. With ITD, the test population is initialized at difficulty 0 and increases incrementally. Table IV shows the performance of the algorithm under different minimal fitness values for members of the test population. The results are averaged over 3 runs of 50000 cycles.

From table IV it is clear that the algorithm with ITD performs best when the minimal fitness value is set to 1. Figure 3 shows the best run with ITD at minimal fitness value 1 over time compared against the best runs without ITD at difficulty 0 and 5.

The CGA with ITD has a slower learning curve. Looking at the benchmark score over time at difficulty 10, it can be seen that the learning curve resembles the one of the CGA at test difficulty 0 at first, but in time shifts to the learning curve of the CGA at test difficulty 5.

Minimal fitness	0	3	5	10
0	3319	1309	1094	882
0.25	3560	1148	943	718
0.5	3597	1412	1176	899
0.75	3537	1324	1149	842
1.0	3375	1675	1401	1086
1.25	2986	1154	999	813
1.5	3242	1317	1070	852

TABLE IV: Results from runs with ITD at different minimal fitness values

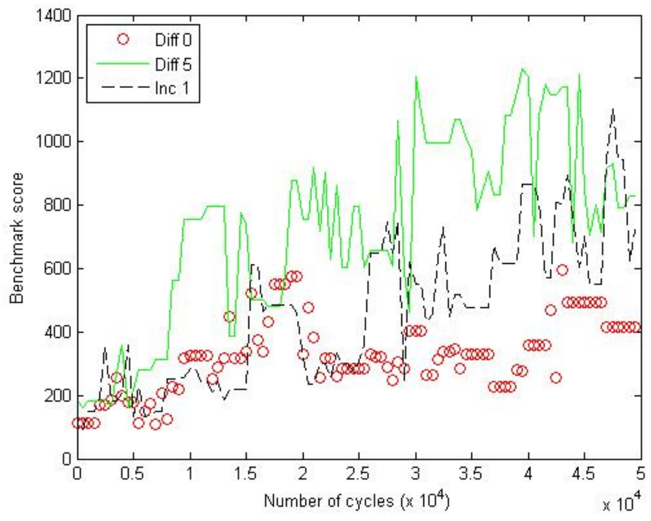

Fig. 3. Run with ITD at minimal fitness 1 (Inc 1) against two runs without ITD, one initialized at difficulty 0 and one at difficulty 5. The benchmark scores are on levels of difficulty 10.

The lack of increase in performance at higher test difficulties explains why runs of the algorithm with ITD enabled do not outperform the runs with ITD disabled. ITD replaces a segment that does not fulfill minimal fitness requirements by a segment of a higher difficulty. Since the results without ITD show that the performance does not rise after a test difficulty of 3, replacing a segment by one of higher difficulty does not improve performance after difficulty 3.

This can be seen by looking at the benchmark scores over time of the algorithm with ITD. With ITD, the algorithm has a slower learning curve but converges to the same optimum as the algorithm without ITD. This is because the average difficulty of the members in the test population is very low at first. The start of the run shows the same learning curve as the regular algorithm initialized at difficulty 0. When the average difficulty increases, the learning curve shows more similarities with the algorithm initialized at higher difficulties.

V. Future Research

For future research, a lot of different aspects of the algorithm can still be tweaked. This paper has only researched the CGA under the settings that seemed fitting. The focus of this paper has mainly been about initial results and finding some of the best parameters. Different population sizes or more specific fitness calculations might yield better results.

While the agents are able to travel larger distances, they still often get stuck or die. They will still sometimes jump blindly into gaps or run into enemies. A large part of the faulty behaviour can be explained by the memoryless property of the networks. Since the agent does not store any data about the previous state of the game, the agent is unable to determine which direction enemies are moving and, more importantly, which direction the agent is moving. For instance, when jumping, the agent needs to know if he is on his way up or down, in order to decide whether he is able to bridge a gap. In many situations, the action the agent needs to make is not just dependent on the game state as is, but also on several previous ones.

Considering the problems resulting from lack of memory, in future work, different types of networks, which incorporate information about previous game states, might produce better results. Since at higher difficulties the amount of elements is very large, scaling the network to perceive more elements might produce better results at higher difficulties.

VI. Conclusions

From the results of the testing, it is clear that the agents developed by the CGA outperform those developed by more traditional algorithms in [12]. The best agents developed by the CGA reach more than twice the distance than similar agents developed in [12]. This is most likely due to a combination of factors. First, the agent is trained on specific segments with a much more specific fitness calculation. The CGA is able to pinpoint the exact points of failure in a level and improve upon specific bottlenecks. By means of partial testing, different sets of behaviour can be isolated and identified more easily. Secondly, the agent gets much more information from the environment due to the different agent architecture. The shifted (floating point) input makes a large difference against the discrete input in [12]. As for the settings, a test population difficulty of 5 and a segment size of 50 blocks produce the best results. The results do not differ significantly from results obtained at a higher test difficulty than 0.

The current research revealed an important characteristic of the game: at difficulty 3 or higher nothing is to be learned anymore. For this reason ITD not really improved the performance. For other games, in which learning extends over more dificulties, ITD might improve performance.

References

[1] David B Fogel. *Blondie24: Playing at the Edge of AI*. Morgan Kaufmann, 2001.

[2] Sergey Karakovskiy and Julian Togelius. The mario ai benchmark and competitions. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):55–67, 2012.

[3] George Konidaris, Dylan Shell, and Nir Oren. Evolving neural networks for the capture game. In *Proceedings of the SAICSIT Postgraduate Symposium*, page 75. Citeseer, 2002.

[4] Chris Miles, Juan Quiroz, Ryan Leigh, and Sushil J Louis. Co-evolving influence map tree based strategy game players. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pages 88–95. IEEE, 2007.

[5] Jan Paredis. Steps towards co-evolutionary classification neural networks. In *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 102–108, 1994.

[6] Jan Paredis. Coevolutionary computation. *Artificial life*, 2(4):355–375, 1995.

[7] Matt Parker and Gary B Parker. The evolution of multi-layer neural networks for the control of xpilot agents. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pages 232–237. IEEE, 2007.

[8] Jordan B Pollack and Alan D Blair. Co-evolution in the successful learning of backgammon strategy. *Machine Learning*, 32(3):225–240, 1998.

[9] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Englewood Cliffs, 2010.

[10] Julian Togelius. Mario ai competetion. http://julian.togelius.com, 2009.

[11] Julian Togelius. 2012 mario ai championship. http://www.marioai.org, 2012.

[12] Julian Togelius, Sergey Karakovskiy, Jan Koutník, and Jürgen Schmidhuber. Super mario evolution. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 156–161. IEEE, 2009.

[13] Author Unknown. Nintendo mini classics super mario bros. http://www.ign.com.