REALM: A Rule-Based Evolutionary Computation Agent that Learns to Play Mario

Slawomir Bojarski and Clare Bates Congdon

Abstract— **REALM is a rule-based evolutionary computation agent for playing a modified version of Super Mario Bros. according to the rules stipulated in the Mario AI Competition held in the 2010 IEEE Symposium on Computational Intelligence and Games. Two alternate representations for the REALM rule sets are reported here, in both hand-coded and learned versions. Results indicate that the second version, with an abstracted action set, tends to perform better overall, but the first version shows a steeper learning curve. In both cases, learning quickly surpasses the hand-coded rule sets.**

I. Introduction

In the realm of video games, Mario is the mascot of the Japanese video game company Nintendo and stars in what many regard as some of the best games in the "platforming" genre. A platforming game is characterized by jumping to and from platforms or over obstacles where the jumps are controlled by the player and it is possible to miss jumps and fall off of the platforms [1]. Platforming games, especially the older ones, are also characterized by being easy to learn, having challenging game-play, and requiring quick reflexes from the player.

Video games are a good environment for artificial intelligence because games contain fully functioning simulated environments that can be used as a testbed for development of artificial intelligence techniques. Recently, some conferences on artificial intelligence have included competitions for creating game-playing agents. One such competition is the Mario AI Competition [2], which was held at the ICE-GIC (London) and CIG (Milan) conferences in 2009 and recently at EvoStar (Istanbul).

The Mario AI Competition is a competition to create the best game-playing agent for a modified version of Super Mario Bros. The top scoring agents from these competitions used the A* search algorithm to guide Mario through the levels. Since the agents that used the A* algorithm did much better than other agents that used other techniques that possibly involved learning, additional tracks were added to the competition for 2010. There is now a game-playing track (similar to the previous competition), a learning track (like the game-playing track, but with learning agents), and a level generation track (create the best agent that generates levels with given characteristics).

Creating agents that learn to play Mario is no easy task, which is why there is a separate track for learning in the Mario AI Competition. The goals of this project were to develop a vocabulary of conditions and actions for a rule-based system and then first, to develop a fixed rule set using this vocabulary and second, to explore the use of evolutionary computation to evolve the rule set in order to perform the task of playing Mario.

The system that is described in this paper is named REALM, which stands for a **r**ule-based **e**volutionary computation **a**gent that **l**earns to play **M**ario. We plan on entering REALM for the learning track of the 2010 competition held at the CIG conference in Copenhagen. In the learning track, the agent is evaluated 10,001 times on a level. The agent's score is the score obtained on the $10{,}001^{\text{th}}$ run. Five different levels are suggested for the agent with the first level being the easiest.

The remainder of this paper proceeds as follows: Section II describes the task and related work; Section III describes both the initial and current designs of REALM, including the vocabulary of conditions and actions; Section IV describes the learning mechanism used by REALM; Section V describes the experiments we ran; Section VI presents the results; Section VII presents conclusions; and Section VIII presents future work.

II. Task Overview and Related Work

This section describes the game used in the Mario AI competition and other related work.

A. Task Overview

The game that is used in the Mario AI competition is a modified version of Markus Persson's Infinite Mario Bros [3]. The objective of the game is to get Mario to the goal which is initially off-screen on the far right. The camera is centered on Mario as he proceeds through a level. Levels may contain enemies, environmental hazards, and helpful items. Whenever Mario gets hit by an enemy, his mode goes down a level. The modes for Mario, in decreasing order, are: fire, large, and small. Mario starts a level in fire mode. A level ends when Mario reaches the goal, runs out of time, falls into a pit, or is hit by an enemy while in small mode. The following are the main enemy types:

- **Goomba**: Can be defeated by stomping on them or by hitting them with a fireball or a shell.
- **Koopa**: Come in two colors: red and green. Red Koopas stick to the platform that they are on while Green Koopas do not. This makes Green Koopas more dangerous than Red Koopas because one can walk off a ledge above Mario and land on top of him. Koopas

Slawomir Bojarski and Clare Bates Congdon are with the Department of Computer Science, University of Southern Maine, Portland, ME 04104 USA (email: slawomir.bojarski@maine.edu, congdon@usm.maine.edu).

978-1-4244-6297-1/10/$26.00 ©2010 IEEE

are vulnerable to the same strategies that Goombas are vulnerable to.
- **Spiky**: Have spiky shells and thus are not vulnerable to stomps. Spikys are also immune to fire. This makes the shell the only weapon that can defeat a Spiky.
- **Bullet Bill**: Bullets that shoot out of cannons and are vulnerable to stomps and shells, but not fire.
- **Piranha Plant**: Jump out of green pipes and are vulnerable to fire and shells, but not stomps.

All enemies can hurt Mario by bumping into him, but each has their own set of vulnerabilities. With the least amount of vulnerabilities, the Spiky would easily be considered the most dangerous enemy that Mario can come across. In addition, there are winged versions of Goombas, Koopas, and Spikys. Winged enemies jump up and down while moving, which makes them harder to hit. Winged Goombas and Winged Koopas lose their wings and revert to their normal types when Mario stomps on them.

Mario may come across the following items:

- **Mushroom**: Change Mario's mode from small to large.
- **Fireflower**: Change Mario's mode to fire regardless of his current mode.
- **Coin**: Increase Mario's in-game score.
- **Koopa shell**: Appear when Mario stomps on a wingless Koopa enemy. Mario can then either stomp on the shell to make it slide or pick up and throw the shell.

Koopa shells defeat any enemy that they come in contact with when sliding, thrown, or simply held by Mario. This makes shells the most powerful weapon at Mario's disposal. There are some drawbacks to using shells. If Mario defeats an enemy with a shell while holding it, the shell disappears. Shells bounce off of scene objects, so Mario has to be careful not to get hit by the shells he uses.

The Mario AI API exposes certain aspects of the environment to agents. The environment that the agent sees is a 22 by 22 grid of the area around Mario. There are two versions of this grid: one containing enemy locations and another containing level scene objects. The float positions of Mario and nearby enemies are given. In addition, the following information on Mario's current state is given: Mario's mode, whether Mario may jump, whether Mario may shoot, whether Mario is on the ground, and whether Mario is carrying a shell. During each game tick, the Mario AI API requests an action from the agent. The agent's action is determined by an array of boolean values where each value represents whether or not a certain key is being pressed. The key presses that are available to Mario are the following:

- **Left**: Moves Mario toward the left of the level.
- **Right**: Moves Mario toward the right of the level.
- **Down**: Makes Mario duck. This also helps slow Mario down when running.
- **Jump**: Makes Mario jump. This is used to get around obstacles and for stomping enemies. Holding jump while in mid-air makes Mario jump higher.
- **Speed**: Makes Mario run faster. When Mario is in fire mode, this is also used to shoot fireballs. This key is also required for executing wall-jumps.

All combinations of key presses are allowed, but some combinations (e.g. Left and Right at the same time) aren't meaningful as they cancel each other out.

B. Related Work

Graphics and physics in video games have been getting better and better with each generation of games. Now that we have realistic looking games in high definition, a natural extension would be to make characters in games behave more naturally [4]. Research in game-playing agents goes back to the 1950s, where traditional board and card games were of interest [5]. Modern games are much more diversified.

Rule-based systems have been used successfully to create agents for arcade games such as Ms. Pac-Man [6], first-person shooters such as Unreal Tournament 2004 [7], racing games such as TORCS [8], and others, but not much work has been done with platforming games [2]. The top-scoring agent of the past few Mario AI Competitions didn't use learning, instead it simulated the environment and used A* to find the optimal jump [9], [10], [11].

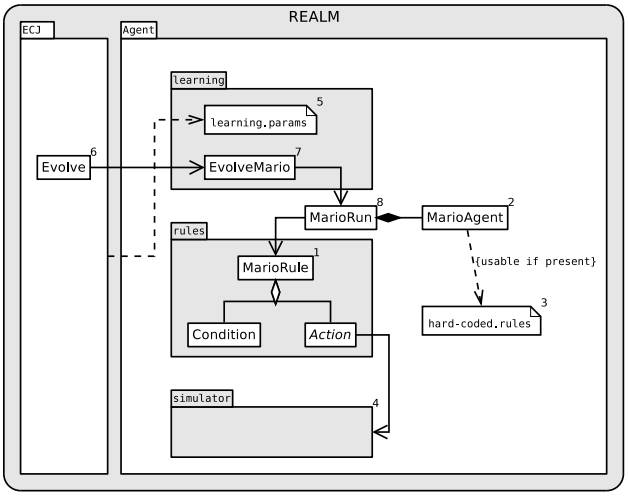


Fig. 1. Structure of REALM v2 in UML notation. The numbers next to items designate the order that they are discussed within the paper.

III. System Design

We chose a rule-based approach for REALM because it is easy to interpret the rules resulting from an evolutionary run and is relatively easy to implement.

After the hard-coded version of REALM was created, we integrated it with the evolutionary computation system ECJ [12] to handle evolution of the rule sets. REALM was coded in the Java programming language, which is the same language the modified version of Super Mario Bros., that the Mario AI Competition uses and the same language used by ECJ.

There are two versions of REALM described in this paper. Both used the the same rule-based architecture, but REALM v2 used fewer conditions than REALM v1, and REALM v1

used key presses as actions while REALM v2 used high-level descriptions of actions.

The following sections describe the rules that we used in both versions of REALM, the agent itself, and our simulation of the environment as used in REALM v2.

A. Rules

Each rule within the rule set contains a vocabulary of conditions along with an associated action (item 1 in Fig. 1). The following describes the conditions and actions used.

1) Conditions: Most of the conditions are ternary-valued; the others are N-ary-valued with N greater than 3. The possible values of a ternary condition are 0, 1, or -1 where they represent FALSE, TRUE, or DONT_CARE, respectively. N-ary conditions assign the same meaning to -1, but allow flexibility in the range of valid values.

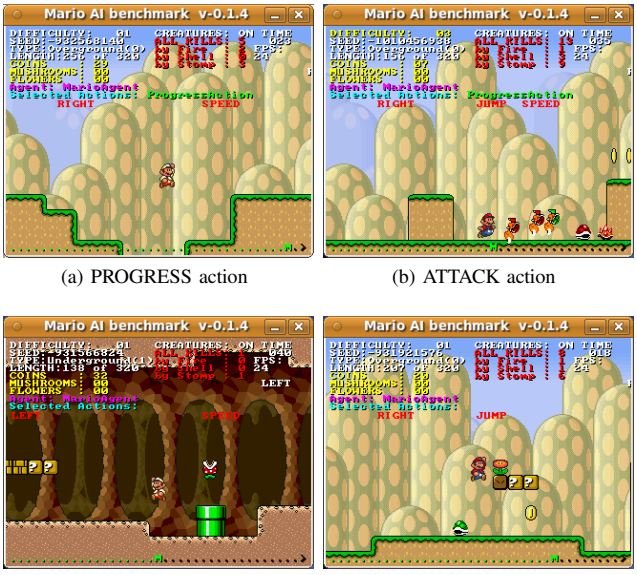
The following is a list of conditions used by REALM v1:

- **MARIO_SIZE**: Represents Mario's status, which can be either small, large, or fire.
- **MAY_MARIO_JUMP**: Represents whether or not Mario is able to jump.
- **IS_MARIO_ON_GROUND**: Represents whether or not Mario is on the ground.
- **IS_MARIO_CARRYING**: Represents whether or not Mario is carrying a Koopa shell.
- **IS_ENEMY_CLOSE_UPPER_LEFT**: Represents the presence of enemies in the upper-left quarter of the environment, within a given range.
- **IS_ENEMY_CLOSE_UPPER_RIGHT**: Represents the presence of enemies in the upper-right quarter of the environment, within a given range.
- **IS_ENEMY_CLOSE_LOWER_LEFT**: Represents the presence of enemies in the lower-left quarter of the environment, within a given range.
- **IS_ENEMY_CLOSE_LOWER_RIGHT**: Represents the presence of enemies in the lower-right quarter of the environment, within a given range.
- **IS_OBSTACLE_AHEAD**: Represents the presence of an obstacle within a given range in front of Mario.
- **IS_PIT_AHEAD**: Represents the presence of a pit within a given range in front of Mario.
- **IS_PIT_BELOW**: Represents the presence of a pit directly below the given location.

The following is a list of conditions used by REALM v2:

- **MARIO_SIZE**: Represents Mario's status, which can be either small, large, or fire.
- **IS_ENEMY_CLOSE_UPPER_LEFT**: Represents the presence of enemies in the upper-left quarter of the environment, within a given range.
- **IS_ENEMY_CLOSE_UPPER_RIGHT**: Represents the presence of enemies in the upper-right quarter of the environment, within a given range.
- **IS_ENEMY_CLOSE_LOWER_LEFT**: Represents the presence of enemies in the lower-left quarter of the environment, within a given range.
- **IS_ENEMY_CLOSE_LOWER_RIGHT**: Represents the presence of enemies in the lower-right quarter of the environment, within a given range.
- **BRICKS_PRESENT**: Represents the presence of bricks within a given range of Mario.
- **POWERUPS_PRESENT**: Represents the presence of power-ups within a given range of Mario.

With the exception of MARIO_SIZE, all the conditions are ternary-valued. MARIO_SIZE has a valid range of values from 0 to 2 (inclusive) where 0, 1, and 2 represent small Mario, large Mario, and fire Mario, respectively.



(a) PROGRESS action (b) ATTACK action

(c) EVADE action (d) POWERUP action

Fig. 2. Mario displaying his actions. In Figure 2a, Mario is using the PROGRESS action to jump over a pit. In Figure 2b, Mario is using the ATTACK action to throw a shell at some enemies. In Figure 2c, Mario is using the EVADE action to evade a Piranha Plant. In Figure 2d, Mario is using the POWERUP action to grab a fireflower item.

2) Actions: The key difference between both versions of REALM is the handling of actions. In REALM v1, actions were composed of distinct key presses. In REALM v2, actions are high-level plans of action or goals for Mario. The following is a list of actions used by REALM v2:

- **PROGRESS**: Represents the objective of reaching the end of a level. This involves Mario moving towards the right side of the screen while maneuvering around obstacles that get in his way such as pits, pipes, cannon towers, steps, uneven ground, and enemies.
- **ATTACK**: Represents the objective of taking the enemy out. There are several ways Mario can dispatch enemies. For instance, if the enemy is not of the spiky variety, he could jump on top of the enemy or, if Mario has fire power, he can shoot fireballs. The last option available to Mario is to either throw or kick the shell of a Koopa enemy. These shells become available when Mario jumps on top of a wingless Koopa. When thrown,

TABLE I

HARD-CODED RULES FOR REALM V1. EMPTY CELLS ARE DON'T CARES FOR CONDITIONS AND FALSE FOR ACTIONS.

#	Conditions											Action				
	Sz	Jmp	Grd	Cry	UL	UR	LL	LR	OAhd	PAhd	PBlw	Left	Right	Down	Jump	Speed
1		1								1			T		T	T
2		1						1					T		T	
3		1							1				T		T	T
4			0						1				T		T	T
5			0								1		T		T	T
6													T			T

shells slide along the ground and take out any enemy that they come in contact with.

- **EVADE**: Represents the objective of evading enemies. Depending on the general direction of the enemies Mario would either find high ground or seek shelter under something. This is useful when Mario can't make it past a barrier of enemies and has to wait, while maintaining a certain distance from enemies, for the right time to either attack or make progress.
- **POWERUP**: Represents the objective of getting power-ups. Mario chases after any power-up within range and hits bricks to force power-ups out. This helps preserve Mario's condition so that Mario can recover from taking damage and go further within a level.

With both versions of REALM using different number of conditions and actions, the number of possible rule combinations are different for each version. REALM v1's rules consisted of 10 ternary-valued conditions, 1 condition with 4 possible values, and an action consisting of 5 boolean values. This would mean that total possible number of rules could be calculated as follows:

$$\text{possible rules} = 4^1 * 3^{10} * 2^5 = 7,558,272$$

With several million possible rules, REALM v1 had to deal with a large search space during evolutionary runs. REALM v2's rules, on the other hand, consisted of 4 fewer ternary-valued conditions and only had 4 actions, which would mean that the total possible number of rules could be calculated as follows:

$$\text{possible rules} = 4^1 * 3^6 * 4 = 11,664$$

With only a little over ten thousand possible rules, REALM v2 has a much smaller search space than REALM v1 had during evolutionary runs. Reduction of the search space was one of the goals in creating REALM v2.

B. Action Selection

Whenever REALM is asked for what action to perform (item 2 in Fig. 1), it first updates its current state and then looks through the rule set to try and find a matching rule. If no matching rule is found, then REALM will use a default plan of action, which is set to be the PROGRESS action. If more than one matching rule is found, then the conflict is resolved by picking the most specific rule first and then by rule order.

C. Hard-coded Rules

Both versions of REALM are capable of loading rules from a text file (item 3 in Fig. 1). Here we describe the rules we created for each version of REALM.

REALM v1 had 6 hard-coded rules (see Table I) that were constructed with the intention of having Mario run toward the right while jumping if enemies, pits, or other obstacles get too close. The first rule instructs Mario to jump to the right while maintaining speed if there is a pit directly ahead (i.e. within 2 grid cells). The second rule instructs Mario to jump to the right if there a close enemy (i.e. within 3 grid cells) in the lower-right quarter of the environment. This includes enemies that are in front of or below Mario. The third rule instructs Mario to jump to the right while maintaining speed if there is an obstacle directly ahead (i.e. within 2 grid cells). The fourth rule instructs Mario to jump higher while trying to jump over an obstacle directly ahead. The fifth rule instructs Mario to jump higher while jumping over pits. The last rule instructs Mario to run to the right regardless of the situation.

TABLE II

HARD-CODED RULES FOR REALM V2. EMPTY CELLS ARE DON'T CARES FOR CONDITIONS.

#	Conditions							Action
	Sz	UL	UR	LL	LR	Brk	Pwr	
1	1	0	0				1	POWERUP
2	1	0	0			1		POWERUP
3	0	0		0			1	POWERUP
4								PROGRESS

REALM v2 had 4 hard-coded rules (see Table II) that were constructed with the intention of having Mario progress through a level while grabbing power-ups on the way when in need of healing. The first two rules instruct Mario to grab a power-up when Mario is large, there are no nearby enemies present in the lower-half of the environment, and either power-ups or bricks are present. The third rule instructs Mario to grab power-ups when Mario is small, there are no nearby enemies present in the lower-half of the environment, and power-ups are present. The reason we chose not to add a rule to grab power-ups when bricks are present, like we did when Mario is large, is due to the fact that Mario cannot break bricks when he is small and we wanted to avoid Mario from getting stuck hitting a regular brick over and over again. The last rule instructs Mario to progress through a level regardless of the situation.

REALM v1 repeated the last action if it didn't find a

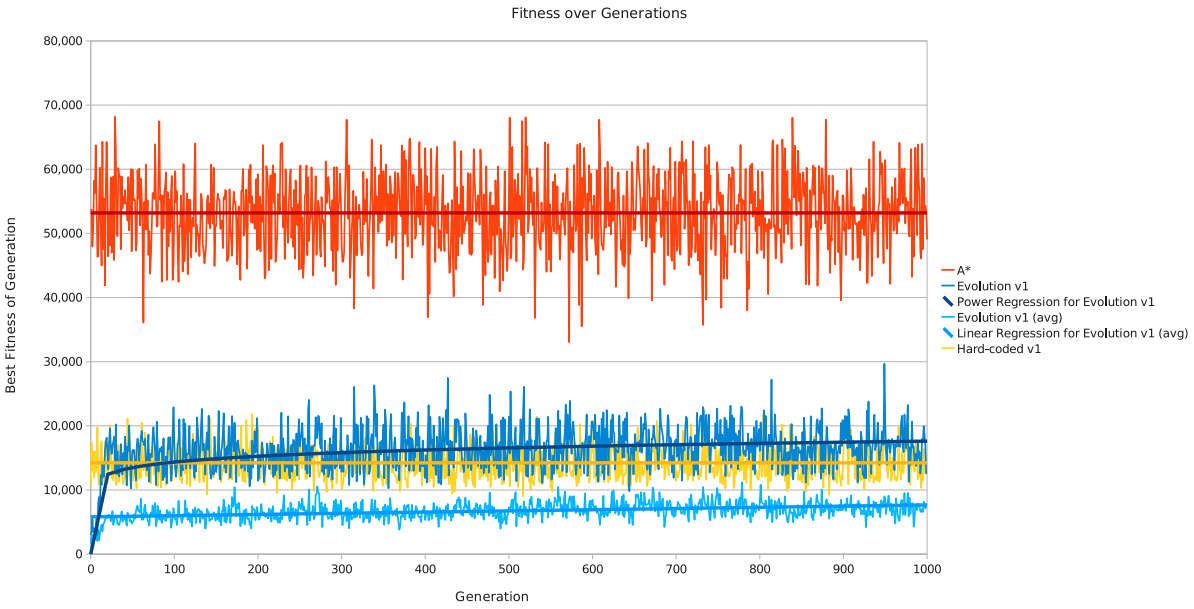


Fig. 3. Performance of REALM v1 and the A* agent during a run through 1,000 generations using a fixed set of seeds

matching rule, so it was important to include a "catch-all" rule (rule #6 in Table I). REALM v2 defaulted to the PROGRESS action, which would mean that rule #4 in Table II is not needed; it is included for clarity.

D. Simulator

Once REALM has selected a plan of action, it picks a goal position based on the plan. Each plan contains a map of weights given to scene objects and enemies. The goal and weight map are passed onto the simulator (item 4 in Fig. 1) which in turn uses them to calculate a path to the goal. Once the simulator obtains a path to the goal, it uses the part of the path that is within Mario's jumping distance to determine the distinct key presses needed to make Mario move along the path. This information is then passed back to the agent.

The simulator uses details on the environment and the weight map to create two separate penalty grids; one for the scene objects and another for the enemies. The simulator then creates a graph of the positions present in the environment and uses the A* algorithm to find a path to the goal position. The A* algorithm uses the penalty grids in determining the scores it assigns to positions.

A special case would be when Mario is inside a pit. When inside a pit, Mario is forced to stick towards a nearby wall in order to set up a wall-jump regardless of the chosen plan. The simulator isn't called during this time. Once Mario is able to wall-jump, the goal is set to the top of the opposite side of the pit and the simulator is called as it normally would be. Because of these wall-jumps and the way the progress action handles approaching pits, REALM is not purely reactive.

IV. Learning

To handle the learning of the rules, we used a Java-based evolutionary computation system called ECJ [12]. ECJ is highly configurable and supports genetic algorithm/programming and Evolutionary Strategies (ES) styles of evolution. Most of the setup with ECJ is done through a configuration file (item 5 in Fig. 1). We used the simple evolution procedure provided by ECJ. The evolution procedure consists of an evaluation phase that evaluates individuals in a population, followed by a breeding phase where parents are selected to produce offspring. The procedure is repeated for a set number of generations. We typically ran around 1,000 generations.

The individuals of a population were configured to contain a single rule set of 20 rules. The size of the population was set to contain 50 individuals. From a fresh start, the rule sets are initialized with random rules. When a rule is randomized, there is a 40% chance that a condition will end up with a value of DONT_CARE.

A. Evaluation Phase

The evaluation phase is where we evaluate an individual by having Mario run through 12 levels (see Table IV) with the individual's rule set and get back an accumulative fitness score for all the levels. The fitness score for a level is computed as follows:

$$\begin{aligned} \text{fitness} = \ & \text{distance} * 1 \\ & + \text{win} * 1024 \\ & + \text{marioMode} * 32 \\ & + \text{timeLeft} * 8 \\ & - \text{died} * (\text{distance} * 0.5) \end{aligned}$$

where $distance$ is how far Mario got in the level; win is 1 when Mario reached the end of a level, 0 otherwise; $marioMode$ is either 0, 1, or 2 for when Mario is small, large, or fire, respectively at the end of a level; $killTotal$ is

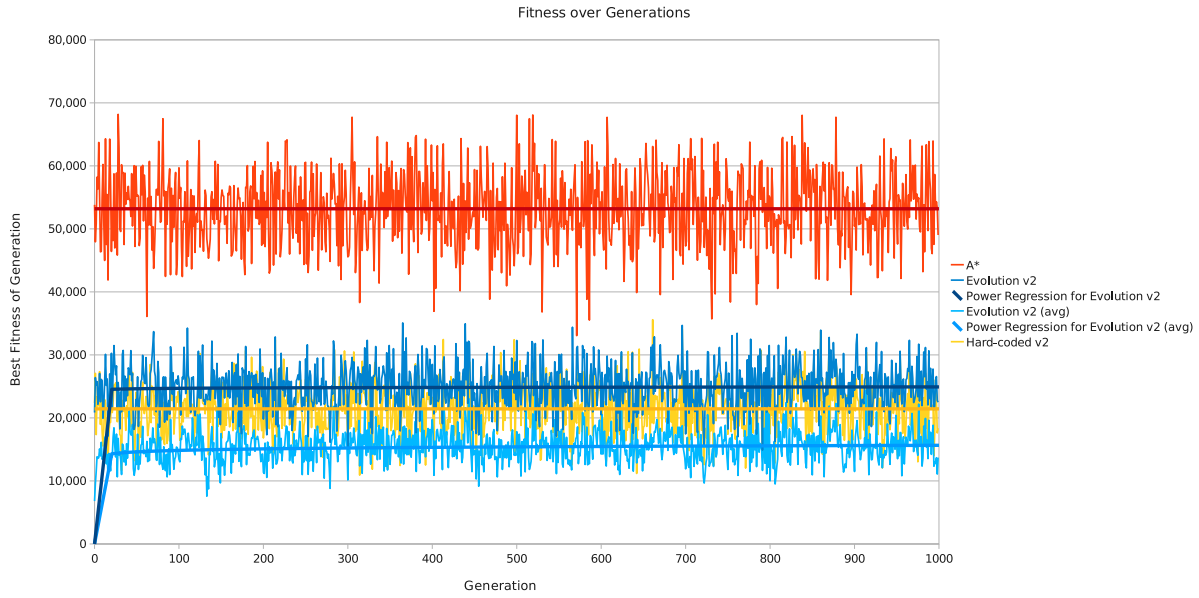


Fig. 4. Performance of REALM v2 and the A* agent during a run through 1,000 generations using a fixed set of seeds

the total number of enemies Mario killed; $killsByStomp$ is the number of enemies Mario killed by jumping on them; $killsByFire$ is the number of enemies Mario killed with fireballs; $killsByShell$ is the number of enemies Mario killed by Koopa shells; and $died$ is 1 when Mario dies in the level, 0 otherwise. The fitness was designed with the intention of having Mario quickly get through a level alive while taking out enemies that would hinder his progress.

B. Breeding Phase

ECJ has a flexible breeding architecture that lets you choose what kind of breeder you want to use as well as how the individuals are selected for breeding. When looking for a breeding strategy, we wanted a strategy that each generation selected the best subset of individuals to be used to breed new rules and also used elitism by keeping those individuals that were used for breeding. Thus we chose to use the $\mu + \lambda$ breeding strategy [13]. μ stands for the number of parents and λ stands for the number of children the parents produce. The $\mu + \lambda$ strategy chooses the μ best individuals as the parents to produce λ children, where the parents are kept in the next generation (elitism) in addition to the children. Each parent produces $\frac{\lambda}{\mu}$ children. λ has to be a multiple of μ and the population size for each generation after the first is fixed at size $\mu + \lambda$. REALM uses 5 for μ and 45 for λ for a population size of 50.

During the breeding procedure, there are some additional operations that are performed with certain probabilities to make the children a little different from their parents. One such operation is mutation, which is where there is a chance for rule conditions and actions to change in value. A second operation is rule crossover, which is where there is a chance of a rule from one child being swapped with another rule from another child. A third operation is rule reordering, which is where there is a chance for a child's rules to be randomly reordered. We set the probabilities for mutation, rule crossover, and rule reordering to 10%, 10%, and 20%, respectively.

Both ECJ and the Mario AI Competition code use random number generators to handle things like, in the case of ECJ, when to execute certain operations and, in the case of the competition code, level generation. To prevent issues such as over fitting [5], we base the seeds for the random number generators off of the current system time. The seed that is used for level generation is recalculated every generation, so that REALM doesn't encounter the same types of levels each generation and learns more generalized rules.

V. Methodology

In order to properly compare REALM to the A* star agent benchmark and previous versions of itself, we had to decide upon a fixed set of seeds to use. Since we can base the seeds off of the current system time, we decided to use the first sets of seeds that were used in an evolutionary run in this fashion.

Runs using hard-coded rules have a different point of entry into REALM than evolutionary runs. During evolutionary runs, the main point of entry is via ECJ's Evolve class (item 6 in Fig. 1). The Evolve class calls REALM's EvolveMario class (item 7 in Fig. 1) for setting up evaluations, performing the actual evaluations, and for printing out various statistics after each evaluation and at the end of evolutionary runs. During evaluations, EvolveMario calls another class called MarioRun (item 8 in Fig. 1) to have the agent run through a series of levels and get back a accumulative fitness score. During regular runs, the main point of entry is the MarioRun class which would evaluate the agent with each seed. This simulates running through multiple generations with a fixed

TABLE III

HIGHEST SCORING RULE SET FROM LAST GENERATION OF EVOLUTION FOR REALM V1. EMPTY CELLS ARE DON'T CARES FOR CONDITIONS AND FALSE FOR ACTIONS.

#	Conditions											Action				
	Sz	Jmp	Grd	Cry	UL	UR	LL	LR	OAhd	PAhd	PBlw	Left	Right	Down	Jump	Speed
1		1	1				0						T		T	T
2	0	0	0	0	1		0	0	0	1	1		T		T	T
3	1	0		1	1	1	0		1	0	0		T		T	T
4	1	0	1		0	0			1		1		T			
5	1	0	1	1	0		0			1	1		T		T	T
6			1	0							0		T			
7	1	0	0			0	1	0	1	1	0		T		T	T
8		0	0			0	0	1	1	1	1		T			
9			1	0	1			0	1	0	0		T		T	T
10	0		1			1	0				1		T			
11		0	0			1	0		0		1		T			
12		0	0		0	0	0	1	0		1		T		T	T
13		1	0	0	0	0	1		0				T		T	T
14	2			1	0	1	1				1		T		T	T
15	1		0	1		1	1	1		0	0		T		T	T
16	1	1	1	1			0	0		1	1		T		T	T
17	1		1	0	1	1	0	1	0	1			T			
18	0	1	1	0	1		0	0	0	1	1		T		T	T
19	1		1			1	0				1		T			
20		0	1	1	0	1	0	0	0	1	1		T		T	T

set of rules.

Each run consists of 12 levels that increase in difficulty and alternate between overground and underground level types. We use the different level types because they are generated differently and thus pose different types of challenges. The accumulative score is set as the fitness for the individual.

TABLE IV

SETTINGS USED FOR EACH RUN

Setting	Values
Level difficulties	1, 3, 5, 12, 16, 20
Level types	0, 1
Creatures enabled	true
Time limits	levelLength * 2 / 10

We ran both versions of REALM for 1,000 generations during evolutionary runs. The hard-coded versions of REALM and the A* agent, that was used as a benchmark, were run for 1,000 repetitions.

VI. RESULTS

After 1,000 generations of learning (see Fig. 4 and Table V), the average obtained by REALM v2 (25,072) was little less than half of what the A* agent obtained (53,190). The difference between REALM v2's best performance (35,078) and the A* agent's worst performance (33,055) was 2,023 points, which is equivalent to the bonus Mario would get for completing two levels.

After 1,000 generations of learning (see Fig. 3 and Table III), the average scores obtained by REALM v1 (16,476) was 65% of REALM v2's average, but was higher than REALM v2's worst performance (14,889). REALM v1's best performance (29,669) was within 5,500 points from REALM v2's best.

Both evolutionary versions of REALM performed better, on average, than their hard-coded counterparts. REALM v1 and REALM v2's average performances were 2,247 and 3,634 points higher than their hard-coded versions.

TABLE V

HIGHEST SCORING RULE SET FROM LAST GENERATION OF EVOLUTION FOR REALM V2. EMPTY CELLS ARE DON'T CARES FOR CONDITIONS.

#	Conditions							Action
	Sz	UL	UR	LL	LR	Brk	Pwr	
1	0	1	0		1	1	0	EVADE
1	0	0		0	1			POWERUP
1	2	1		1	1	1	0	EVADE
1	2		0	0	1			ATTACK
1	2		1		0			EVADE
1	1	1	0	1	0	1	0	EVADE
1		1	1	1	1	1	1	ATTACK
1	0	1	1	0	1	0	1	EVADE
1	2	0		0	1	1		ATTACK
1	2			0	1	0	0	PROGRESS
1	1	1			0		0	EVADE
1	0			1	1	0		ATTACK
1	1	0		0	1		1	POWERUP
1		1	1	0	1	1	1	POWERUP
1	1	1	0	1		1	0	EVADE
1	2	0	0				1	PROGRESS
1	0	0	1	0	0	1	0	EVADE
1	0			1	1			ATTACK
1	0	0		1		1	1	EVADE
1	0	0		1	1	1		PROGRESS

Both versions of REALM improve over time, but at very different rates. REALM v2's best score for each generation seems to barely improve at all, but steady improvement is in fact displayed in the average score of each generation as shown in Fig. 4. REALM v1 shows steady improvement as seen in Fig. 3, but was showing signs of plateauing due to it trending lower after 1,000 generations than what was expected from extrapolating from the first 500 data points.

The competition runs the game at 24 frames per second, which would allow about $40ms$ each frame for the agent

to return an action. REALM had low computational times when compared to the A* agent, which took most of the of $40ms$ to return an action. For calculating the time REALM took to return an action, we used the best rule set from the evolutionary run as a fixed rule set to run through all the same levels again. This resulted in 249,414 actions being recorded. The fastest measured time was $0.0024ms$, the slowest was $50.6755ms$, the average was $0.1058ms$, and the median was $0.0799ms$. The processor used when calculating these times was an Intel Core 2 Duo E8400 and there was 4GB of DDR2 800 RAM. REALM easily meets these time restrictions, with only a few exceptions. Out of the 249,414 recorded actions, only 5 took $40ms$ or longer to compute.

VII. Conclusion

The A* agent uses a simulator that is based off of the game engine, which enables it to know the exact positions of Mario and enemies and thus achieve better performance. REALM, on the other hand, doesn't try to calculate exact locations nor does it look at future positions of enemies yet still achieves competitive levels of performance.

REALM v2 displayed better overall performance than REALM v1, but displayed a less interesting learning curve. In REALM v2, most of the logic which decided the distinct key presses were pushed into the actions, which weren't being evolved. During evolution, REALM v2 dealt with a much smaller search than REALM v1.

REALM v1 displayed a much more interesting learning curve than REALM v2, but its performance suffered due to not being able to handle enemies and pits properly.

Out of both versions of REALM, we have found that REALM v2 was more interesting to watch play the game. REALM v2 displayed more advanced game-play techniques than REALM v1, but still made mistakes due to its imprecise nature. This combination of skill and imperfection made it seem more human-like than either REALM v1 and the A* agent.

The time REALM takes to compute an action is relatively low. This means that we have room to improve the precision of the simulator and still be within the restricted amount of time.

VIII. Future Work

To further improve REALM, we plan to add a cutoff for computation if REALM takes too long in picking an action. We also plan to improve the precision of the simulator by adding additional dimensions to the graph such as vertical and horizontal velocity. Precision can also be improved by trying to predict the future locations of enemies. For this we plan to add "areas of influence" that expand the coverage of penalties associated with an enemy for each simulated time step. This would require less computational time than trying to predict the exact location of an enemy.

In addition, we plan on keeping track of which actions are required between each step in the simulated path, thus we would no longer need to decide on the distinct key presses based on the part of the path that is within Mario's jumping distance.

Fig. 5. Mario trying to wall-jump out of a pit

Acknowledgments

The authors would like to thank Ryan Small, whose Agent Smith project provided inspiration for this project. The authors would also like to thank Kyle Kilgour for giving ideas for the agent and Johnicholas Hines for ideas as well as insight into useful programs for managing the project. Last, but not least, the authors would like to thank Slawomir's sister, Joanna Bojarska, for proofreading an early draft of this paper.

References

[1] Platform game. [Online]. Available: http://en.wikipedia.org/wiki/Platform_game

[2] J. Togelius, S. Karakovskiy, J. Koutník, and J. Schmidhuber, "Super mario evolution," in *CIG'09: Proceedings of the 5th international conference on Computational Intelligence and Games*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 156–161.

[3] Infinite mario bros. [Online]. Available: http://www.mojang.com/notch/mario/index.html

[4] F. Dignum, J. Westra, W. van Doesburg, and M. Harbers, "Games and agents: Designing intelligent gameplay," *International Journal of Computer Games Technology*, vol. 2009, 2009.

[5] L. Galway, D. Charles, and M. Black, "Machine learning in digital games: a survey," *Artif. Intell. Rev.*, vol. 29, no. 2, pp. 123–161, 2008.

[6] A. Fitzgerald and C. B. Congdon, "Ramp: a rule-based agent for ms. pac-man," in *CEC'09: Proceedings of the Eleventh conference on Congress on Evolutionary Computation*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 2646–2653.

[7] R. Small and C. B. Congdon, "Agent smith: towards an evolutionary rule-based agent for interactive dynamic games," in *CEC'09: Proceedings of the Eleventh conference on Congress on Evolutionary Computation*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 660–666.

[8] D. Loiacono, J. Togelius, P. L. Lanzi, L. Kinnaird-heether, S. M. Lucas, M. Simmerson, D. Perez, R. G. Reynolds, and Y. Saez, "The wcci 2008 simulated car racing competition," 2008.

[9] Gic2009competition. [Online]. Available: http://julian.togelius.com/mariocompetition2009/GIC2009Competition.pdf

[10] Cig2009competition. [Online]. Available: http://julian.togelius.com/mariocompetition2009/CIG2009Competition.pdf

[11] Infinite super mario ai. [Online]. Available: http://www.doc.ic.ac.uk/~rb1006/projects:marioai

[12] Ecj. [Online]. Available: http://cs.gmu.edu/~eclab/projects/ecj/

[13] Tutorial 3: Build a floating-point evolution strategies problem. [Online]. Available: http://cs.gmu.edu/~eclab/projects/ecj/docs/tutorials/tutorial3/index.html