The 2009 Mario AI Competition

Julian Togelius, Sergey Karakovskiy and Robin Baumgarten

***Abstract*— This paper describes the 2009 Mario AI Competition, which was run in association with the IEEE Games Innovation Conference and the IEEE Symposium on Computational Intelligence and Games. The focus of the competition was on developing controllers that could play a version of *Super Mario Bros* as well as possible. We describe the motivations for holding this competition, the challenges associated with developing artificial intelligence for platform games, the software and API developed for the competition, the competition rules and organization, the submitted controllers and the results. We conclude the paper by discussing what the outcomes of the competition can teach us both about developing platform game AI and about organizing game AI competitions. The first two authors are the organizers of the competition, while the third author is the winner of the competition.**

Keywords: Mario, platform games, competitions, A*, evolutionary algorithms

I. INTRODUCTION

For the past several years, a number of game-related competitions have been organized in conjunction with major international conferences on computational intelligence (CI) and artificial intelligence for games (Game AI). In these competitions, competitors are invited to submit their best controllers or strategies for a particular game; the controllers are then ranked based on how well they play the game, alone or in competition with other controllers. The competitions are based on popular board games (such as Go and Othello) or video games (such as Pac-Man, Unreal Tournament and various car racing games [1], [2]).

In most of these competitions, competitors submit controllers that interface to an API built by the organizers of the competition. The winner of the competition becomes the person or team that submitted the controller that played the game best, either on its own (for single-player games such as Pac-Man) or against others (in adversarial games such as Go). Usually, prizes of a few hundred US dollars are associated with each competition, and a certificate is always awarded. There is no requirement that the submitted controllers be based on any particular type of algorithm, but in many cases the winners turn out to include computational intelligence (typically neural networks and/or evolutionary computation) in one form or another. The submitting teams tend to comprise both students, faculty members and persons not currently in academia (e.g. working as software developers).

There are several reasons for holding such competitions as part of the regular events organized by the computational intelligence community. A main motivation is to improve benchmarking of learning and other AI algorithms. Benchmarking is frequently done using very simple testbed problems, that might or might not capture the complexity of real-world problems. When researchers report results on more complex problems, the technical complexities of accessing, running and interfacing to the benchmarking software might prevent independent validation of and comparison with the published results. Here, competitions have the role of providing software, interfaces and scoring procedures to fairly and independently evaluate competing algorithms and development methodologies.

Another strong incentive for running these competitions is that they motivate researchers. Existing algorithms get applied to new areas, and the effort needed to participate in a competition is (or at least, should be) less than it takes to write new experimental software, do experiments and write a completely new paper. Competitions might even bring new researchers into the computational intelligence fields, both academics and non-academics. One of the reasons for this is that game-based competitions simply look cool.

The particular competition described in this paper is similar in aims and organization to some other game-related competitions (in particular the simulated car racing competitions) but differs in that it is built on a platform game, and thus relates to the particular challenges an agent faces while playing such games.

A. AI for platform games

Platform games can be defined as games where the player controls a character/avatar, usually with humanoid form, in an environment characterized by differences in altitude between surfaces ("platforms") interspersed by holes/gaps. The character can typically move horizontally (walk) and jump, and sometimes perform other actions as well; the game world features gravity, meaning that it is seldom straightforward to negotiate large gaps or altitude differences.

To our best knowledge, there have not been any previous competitions focusing on platform game AI. The only published paper on AI for platform games we know of is a recent paper by the first two authors of the current paper, where we described experiments in evolving neural network controllers for the same game as was used in the competition, using an earlier version of the API [3]. Some other papers have described uses of AI techniques for automatic generation of levels for platform games [4], [5], [6].

Most commercial platform games incorporate little or no AI. The main reason for this is probably that most platform games are not adversarial; a single player controls a single character who makes its way through a sequence of levels, with his success dependent only on the player's skill. The

JT is with IT University of Copenhagen, Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark. SK is with IDSIA, Galleria 2, 6928 Manno-Lugano, Switzerland. RB is with Imperial College of Science and Technology, London, United Kingdom. Emails: julian@togelius.com, sergey@idsia.ch, robin.baumgarten06@doc.ic.ac.uk

obstacles that have to be overcome typically revolve around the environment (gaps to be jumped over, items to be found etc) and NPC enemies; however, in most platform games these enemies move according to preset patterns or simple homing behaviours.

Though apparently an under-studied topic, artificial intelligence for controlling the player character in platform games is certainly not without interest. From a game development perspective, it would be valuable to be able to automatically create controllers that play in the style of particular human players. This could be used both to guide players when they get stuck (cf. Nintendo's recent "Demo Play" feature, introduced to cope with the increasingly diverse demographic distribution of players) and to automatically test new game levels and features as part of an algorithm to automatically tune or create content for a platform game.

From an AI and reinforcement learning perspective, platform games represent interesting challenges as they have high-dimensional state and observation spaces and relatively high-dimensional action spaces, and require the execution of different skills in sequence. Further, they can be made into good testbeds as they can typically be executed much faster than real time and tuned to different difficulty levels. We will go into more detail on this in the next section, where we describe the specific platform game used in this competition.

II. Infinite Mario Bros

The competition uses a modified version of Markus Persson's *Infinite Mario Bros*, which is a public domain clone of Nintendo's classic platform game *Super Mario Bros*. The original Infinite Mario Bros is playable on the web, where Java source code is also available[1].

The gameplay in Super Mario Bros consists in moving the player-controlled character, Mario, through two-dimensional levels, which are viewed sideways. Mario can walk and run to the right and left, jump, and (depending on which state he is in) shoot fireballs. Gravity acts on Mario, making it necessary to jump over holes to get past them. Mario can be in one of three states: *Small*, *Big* (can crush some objects by jumping into them from below), and *Fire* (can shoot fireballs).

The main goal of each level is to get to the end of the level, which means traversing it from left to right. Auxiliary goals include collecting as many as possible of the coins that are scattered around the level, finishing the level as fast as possible, and collecting the highest score, which in part depends on number of collected coins and killed enemies.

Complicating matters is the presence of holes and moving enemies. If Mario falls down a hole, he loses a life. If he touches an enemy, he gets hurt; this means losing a life if he is currently in the Small state. Otherwise, his state degrades from Fire to Big or from Big to Small. However, if he jumps and lands on an enemy, different things could happen. Most enemies (e.g. goombas, cannon balls) die from this treatment; others (e.g. piranha plants) are not vulnerable to this and proceed to hurt Mario; finally, turtles withdraw into their shells if jumped on, and these shells can then be picked up by Mario and thrown at other enemies to kill them.

Certain items are scattered around the levels, either out in the open, or hidden inside blocks of brick and only appearing when Mario jumps at these blocks from below so that he smashes his head into them. Available items include coins, mushrooms which make Mario grow Big, and flowers which make Mario turn into the Fire state if he is already Big.

A. Automatic level generation

While implementing most features of Super Mario Bros, the standout feature of Infinite Mario Bros is the automatic generation of levels. Every time a new game is started, levels are randomly generated by traversing a fixed width and adding features (such as blocks, gaps and opponents) according to certain heuristics. The level generation can be parameterized, including the desired difficulty of the level, which affects the number and placement of holes, enemies and obstacles. In our modified version of Infinite Mario Bros we can specify the random seed of the level generator, making sure that any particular level can be recreated by simply using the same see. Unfortunately, the current level generation algorithm is somewhat limited; for example, it cannot produce levels that include dead ends, which would require back-tracking to get out of.

B. Challenges for AIs (and humans)

Several features make Super Mario Bros particularly interesting from an AI perspective. The most important of these is the potentially very rich and high-dimensional environment representation. When a human player plays the game, he views a small part of the current level from the side, with the screen centered on Mario. Still, this view often includes dozens of objects such as brick blocks, enemies and collectable items. The static environment (grass, pipes, brick blocks etc.) and the coins are laid out in a grid (of which the standard screen covers approximately $22 * 22$ cells), whereas moving items (most enemies, as well as the mushroom power-ups) move continuously at pixel resolution.

The action space, while discrete, is also rather large. In the original Nintendo game, the player controls Mario with a D-pad (up, down, right, left) and two buttons (A, B). The A button initiates a jump (the height of the jump is determined partly by how long it is pressed); the B button initiates running mode and, if Mario is in the Fire state, shoots a fireball. Disregarding the unused up direction, this means that the information to be supplied by the controller at each time step is five bits, yielding $2^5 = 32$ possible actions, though some of these are nonsensical (e.g. left together with right).

Another interesting feature is that there is a smooth learning curve between levels, both in terms of which behaviours are necessary and their required degree of refinement. For example, to complete a very simple Mario level (with no enemies and only small and few holes and obstacles) it might be enough to keep walking right and jumping whenever there is something (hole or obstacle) immediately in front of Mario. A controller that does this should be easy to

[1] http://www.mojang.com/notch/mario/

```
// always the same dimensionality 22x22
// always centered on the agent
public byte[][] getCompleteObservation();
public byte[][] getEnemiesObservation();
public byte[][] getLevelSceneObservation();
public float[] getMarioFloatPos();
public float[] getEnemiesFloatPos();
public boolean isMarioOnGround();
public boolean mayMarioJump();
```

Fig. 1. The *Environment* Java interface, which contains the observation, i.e the information the controller can use to decide which action to take.

learn. To complete the same level while collecting as many as possible of the coins present on the same level likely demands some planning skills, such as smashing a power-up block to retrieve a mushroom that makes Mario Big so that he can retrieve the coins hidden behind a brick block, and jumping up on a platform to collect the coins there and then going back to collect the coins hidden under it. More advanced levels, including most of those in the original Super Mario Bros game, require a varied behaviour repertoire just to complete. These levels might include concentrations of enemies of different kinds which can only be passed by timing Mario's passage precisely; arrangements of holes and platforms that require complicated sequences of jumps to pass; dead ends that require backtracking; and so on.

III. Competition API

In order to be able to run the competition (and in order to use the game for other experiments), the organizers modified Infinite Mario Bros rather heavily and constructed an API that would enable it to be easily interfaced to learning algorithms and competitors' controllers. The modifications included removing the real-time element of the game so that it can be "stepped" forward by the learning algorithm, removing the dependency on graphical output, and substantial refactoring (the developer of the game did not anticipate that the game would be turned into an RL benchmark). Each time step, which corresponds to 40 milliseconds of simulated time (an update frequency of 25 fps), the controller receives a description of the environment, and outputs an action. The resulting software is a single-threaded Java application that can easily be run on any major hardware architecture and operating system, with the key methods that a controller needs to implement specified in a single Java interface file (see figure 1). On an iMac from 2007, $5-20$ full levels can be played per second (several thousand times faster than real-time). A TCP interface for controllers is also provided.

Figure 2 shows the size and layout relative to Mario of the sensor grid. For each grid cell, the controller can choose to read the complete observation (using the *getCompleteObservation()* method) that returns an integer value representing exactly what occupies that part of the environment (e.g. a section of a pipe), or one of the simplified observations (*getEnemiesObservation()* and *getLevelSceneObservation()*) that simply returns a zero or one signifying the presence of an enemy or an impassable part of the environment.

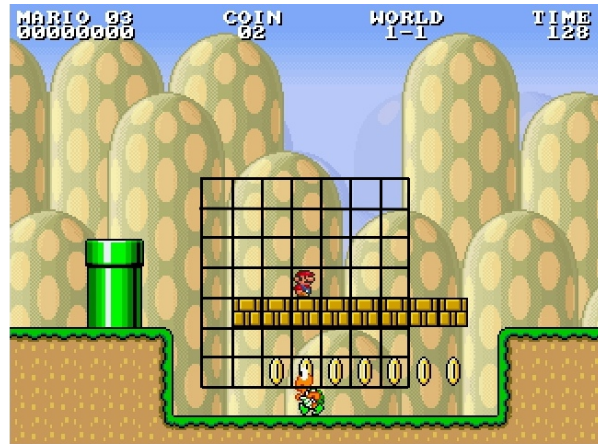

Fig. 2. Visualization of the granularity of the sensor grid. The top six environment sensors would output 0 and the lower three input 1. All of the enemy sensors would output 0, as even if all 49 enemy sensors were consulted none of them would reach all the way to the body of the turtle, which is four blocks below Mario. None of the simplified observations register the coins, but the complete observation would. Note that larger sensor grids can be used (up to 22×22), if the controller can handle the information.

IV. Competition organization and rules

The organization and rules of the competition sought to fulfill the following objectives:

1) *Ease of participation.* We wanted researchers of different kinds and of different levels of experience to be able to participate, students as well as withered professors who haven't written much code in a decade.
2) *Transparency.* We wanted as much as possible to be publicly known about the competing entries, the benchmark software, the organization of the competition etc. This can be seen as an end in itself, but a more transparent competition also makes it easier to detect cheaters, to exploit the scientific results of the competition and to reuse the software developed for it.
3) *Ease of finding a winner.* We wanted it to be unambiguously clear how to rank all submitted entries and who won the competition.
4) *Depth of challenge.* We wanted there to be a real score difference between controllers of different quality, both at the top and the bottom of the high-score table.

The competition web page hosts the rules, the downloadable software and the final results of the competition[2]. Additionally, a Google Group was set up to which all technical and rules questions were to be posted, so that they came to the attention of and could be answered by both organizers and other competitors[3], and where the organizers posted news about the competition. The searchable archive of the discussion group functions as a repository of technical information about the competition.

Competitors were free to submit controllers written in any programming language and using any development method-

[2]http://julian.togelius.com/mariocompetition2009
[3]http://groups.google.com/group/marioai

ology, as long as they could interface to an unmodified version of the competition software and control Mario in real time on a standard desktop PC running Mac OS X or Windows XP. For competitors using only Java, there was a standardized submission format. Any submission that didn't follow this format needed to be accompanied by detailed instructions for how to run it. Additionally, the submission needed to be accompanied by the score the competitors had recorded themselves using the *CompetitionScore* class that was part of the competition software, so that the organizers could verify that the submission ran as intended on their systems. We also urged each competitor to submit a description of how the controller works as a text file.

Competitors were urged to submit their controllers early, and then re-submit improved versions. This was so that any problems that would have disqualified the controllers could be detected and rectified and the controllers resubmitted. No submissions or resubmissions at all were accepted after the deadline (about a week before each competition event). The most common problems that mandated resubmission were:

- The controller used a nonstandard submission format, and no running instructions were provided.
- The controller was submitted together with and entangled in a complete version of the competition software.
- The controller timed out, i.e. took more than 40 ms per time step *on average* on some level.

A. Scoring

All entries were scored before the conference through running them on 10 levels of increasing difficulty, and using the total distance travelled on these levels as the score. The scoring procedure was deterministic, as the same random seed was used for all controllers, except in the few cases where the controller was nondeterministic. The CompetitionScore method uses a supplied random number seed to generate the levels. Competitors were asked to score their own submissions with seed 0 so that this score could be verified by the organizers, but the seed used for the competition scoring was not generated until after the submission deadline, so that competitors could not optimize their controllers for beating a particular sequence of levels.

For the second phase of the competition (the CIG phase) we discovered some time before the submission deadline that two of the submitted controllers were able to clear all levels for some random seeds. We therefore modified the *CompetitionScore* class so as to make it possible to differentiate better between high-scoring controllers. First of all, we increased the number of levels to 40, and varied the length of the levels stochastically, so that controllers could not be optimized for a fixed level length. In case two controllers still cleared all 40 levels, we defined three tie-breakers: game-time (not clock-time) left at the end of all 40 levels, number of total kills, and mode sum (the sum of all Mario modes at the end of levels, where 0=small, 1=big and 2=fire; a high mode sum indicates that the player has taken little damage). So if two controllers both cleared all levels, that one that took the least time to do so would win, and if both took the same time the most efficient killer would win etc.

V. Media strategy and coverage

The competition got off of to a slow start. The organizers did not put a final version of the competition software online until relatively late, and neglected advertising the competition until about a month and a half before the deadline of the first phase. Once it occurred to the organizers to advertise their competition, they first tried the usual channels, mainly mailing lists for academics working within computational intelligence. One of the organizers then posted a link to the competition on his *Facebook* profile, which led one of his friends to submit the link to the social media site *Digg*. This link was instantly voted up on the front page of the site, and a vigorous discussion ensued on the link's comment page. This also meant 20000 visitors to the competition website in the first day, some of which joined the discussion group and announced their intention to participate. Seeing the success of this, the organizers submitted a story about the competition to the technology news site *Slashdot*. Slashdot is regularly read by mainstream and technology news media, which meant that within days there were stories about the competition in (among others) *New Scientist*[4], *MSNBC*[5] and *Le Monde*[6].

At this point, Robin Baumgarten had already finished a first version of his controller, and posted a video on YouTube of his controller guiding Mario through a level, complete with a visualization of the alternate path the A* algorithm were evaluating at each time step[7]. The video was apparently so enjoyable to watch that it quickly reached the top lists, and has at the time of writing been viewed over 600000 times. The success of this video was a bit of a double-edged sword; it drew hordes of people to the competition website, and led to several announcements of intention to participate (though far from everybody who said they would eventually submitted a controller), but it also caused some competitors to drop out as Robin's controller was so impressive that they considered themselves to not stand a chance to win.

VI. Submitted controllers

This section describes the controllers that were submitted to the competition. We describe the versions of the controllers that were submitted to the CIG phase of the competition; in several cases, preliminary versions were submitted to the ICE-GIC phase. The winning controller is described in some detail, whereas the other controllers are given shorter descriptions due to space consideration. In some cases, a controller is either very similar to an existing controller or very little information was provided by the competitor; those controllers are described only very briefly here.

[4]http://www.newscientist.com/article/dn17560-race-is-on-to-evolve-the-ultimate-mario.html

[5]http://www.msnbc.msn.com/id/32451009/ns/technology_and_science-science/

[6]http://www.lemonde.fr/technologies/article/2009/08/07/quand-c-est-l-ordinateur-qui-joue-a-mario_1226413_651865.html

[7]http://www.youtube.com/watch?v=DlkMs4ZHHr8

A. Robin Baumgarten

As the goal of the competition was to complete as many levels as possible, as fast as possible, without regarding points or other bonuses, the problem can be seen as a path optimisation problem: What is the quickest route through the environment that doesn't get Mario killed? *A* search* is a well known, simple and fast algorithm to find shortest paths which seemed perfect for the Mario competition. As it turned out, A* search is fast and flexible enough to find routes through the generated levels in real-time.

The implementation process can be separated into three phases: (a) building a physics simulation including world states and object movement, (b) using this simulation in an A* planning algorithm, and (c) optimising the search engine to fulfill real-time requirements with partial information.

a) Simulating the Game Physics: Due to being open-source, the entire physics engine of *Infinite Mario Bros* is directly accessible and can be used to simulate future world states that correlate very closely with the actual future world state. This was achieved by copying the entire physics engine to the controller and removing all unnecessary parts, such as rendering and some navigation code.

Associating simulated states of enemies with the received coordinates from the API provided an implementation challenge. This association is needed because the API provides only the location and not the current speed, which has to be derived from consecutive recordings. Without accurate enemy behaviour simulation, this can be difficult, especially if there are several enemies in a close area.

b) A for Infinite Mario:* The *A* search algorithm* is a widely used best-first graph search algorithm that finds a path with the lowest cost between a pre-defined start node and one out of possibly several goal-nodes[7]. A* uses a heuristic that estimates the remaining distance to the goal nodes in order to determine the sequence of nodes that are searched by the algorithm. It does so by adding the estimated remaining distance to the previous path cost from the start node to the current node. This heuristic should be *admissible* (not overestimate this distance) for optimality to be guaranteed. However, if only near-optimality is required, this constraint can be relaxed to speed up path-finding. With a heuristic that overestimates by x, the path will be at most x too long[8].

To implement an A* search for Infinite Mario, we have to define what a *node* and a *neighbour of a node* is, which nodes are goal nodes, and how the heuristic estimates the remaining distance.

A node is defined by the entire world-state at a given moment in time, mainly characterised through Mario's position, speed, and state. Furthermore, (re)movable objects in the environment have to be taken into account, such as enemies, power-ups, coins and boxes. The only influence we have on this environment is interaction through actions that Mario can perform. These *Mario actions* consist of combinations of movements such as *left*, *right*, *duck*, *jump* and *fire*.

Neighbours of a node are given by the world state after one (further) simulation step, applying an action that Mario executes during this simulation step. See figures 3 and 4 for an illustration of a typical scenario, figure 5 for an in-game visualization. Note how backtracking takes place when an obstacle is discovered, and how the location of the subsequent neighbours is influenced by the current speed of Mario in figure 4.

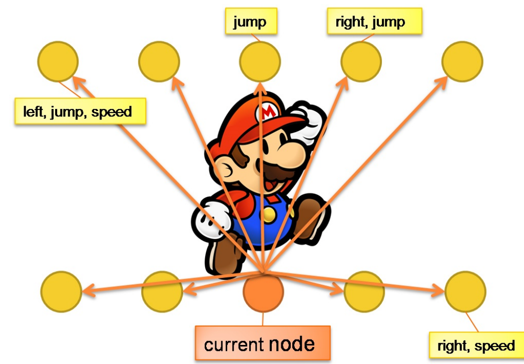

Fig. 3. Illustration of node generation in A* search. Each neighbour node in the search is given by a combination of possible input commands.

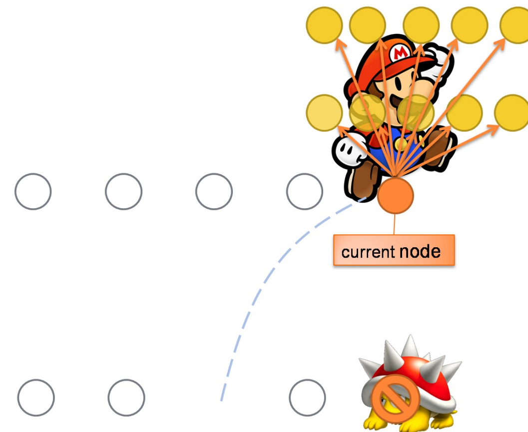

Fig. 4. Illustration of node placement in A* search in Infinite Mario. The best node in the previous simulation step (right, speed) is inaccessible because it contains an obstacle. Thus the next best node is chosen, which results in a jump over the obstacle. The search process continues by generating all possible neighbours. The speed of Mario at the current node distorts the positions of all following nodes.

As only a small window of the world around Mario is visible in the competition API, the level structure outside of this window is unknown. Therefore it does not make sense to plan further ahead than the edge of the current window. Thus, a goal node for our planner is defined as any node where Mario is outside (or just before) the right border of the window of the known environment.

In the standard implementation of A*, the heuristic estimates the remaining distance to the target. In Infinite Mario, one could just translate this as the horizontal distance to the right border of the window. However, this does not lead to a very accurate heuristic, as it ignores the current speed of Mario. To make sure the bot has an admissible (i.e., underestimating) heuristic, it would have to assume Mario runs with maximum speed towards the goal, which is often

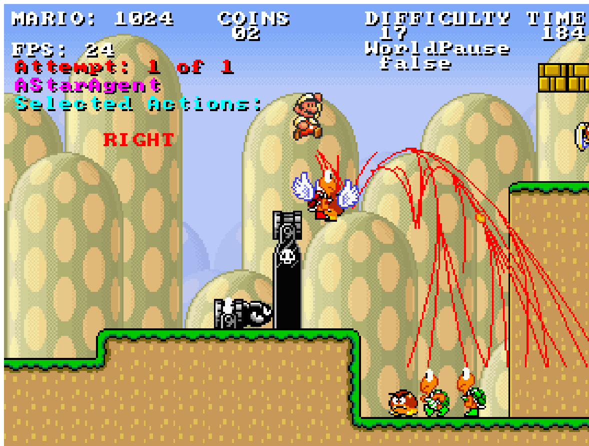

Fig. 5. Visualization of the future paths considered by the A* controller. Each read line shows a possible future trajectory for Mario, taking the dynamic nature of the world into account.

not the case. Instead, a more accurate representation of the distance to the goal can be given by simulating the time required to reach the right border of the window. Here, the current speed of Mario is taken into account. The quickest solution to get to the goal would then be to accelerate as fast as possible, and taking the required time as a heuristic. Similarly, the previous distance of a node to the starting node is simply the time it took to reach the current node.

c) Variable Optimisation: While the A* search algorithm is quite solid and guarantees optimality, certain restrictions need to be put on its execution time to stay within the allowed 40ms for each game update. These restrictions will likely lead to a non-optimal solution, so careful testing has to be undertaken to ensure that the search terminates.

The first restriction used was to stop looking for solutions when the time-limit had been reached, and using the node with the best heuristic so far as a goal node. The linear level design created by the level generator in Infinite Mario favors this approach, as it does not feature dead ends that would require back-tracking of suboptimal paths.

The requirement that the heuristic has to be admissible has also been relaxed. A slightly overestimating heuristic increases the speed of the algorithm in expense of accuracy[8]. In detail, the estimation of the remaining distance can be multiplied with a factor $w >= 1$. Experimentation with different values for w, led to an optimal factor of $w = 1.11$.

Another factor that has an effect on the processing time required by A* is the frequency of plan recalculation. As mentioned above, limited information requires a recalculation of the plan once new information becomes available, i.e., obstacles or enemies enter the window of visible information. Experimentation indicated that the best balance between planning ahead and restarting the planning process to incorporate new information is given when the plan is recreated every two game updates. Planning longer in advance occasionally led to reacting too late to previously unknown threats, resulting in a lost life or slowdown of Mario.

B. Other A*-based controllers

Two other controllers were submitted that were based on the A* algorithm. One was submitted by *Peter Lawford* and the other by a team consisting of *Andy Sloane, Caleb Anderson and Peter Burns* (we will refer to this team with Andy's name in the score tables). These submission were inspired by Robin's controller and used a similar overall approach, but differed significantly in implementation.

C. Other hand-coded controllers

The majority of submitted controllers were hand-coded and did (to our best knowledge) not use any learning algorithms, nor much internal simulation of the environment. Most of these continuously run rightwards but use heuristics to decide when and how to jump. Some were built on one of the standard heuristic controllers supplied with the competition software. The following are very (due to space limitations) brief characterizations of each:

1) Trond Ellingsen: Rule-based controller. Determines a "danger level" of each gap or enemy and acts based on this.

2) Sergio Lopez: Rule-based. Answers the questions "should I jump?" and "which type of jump?" heuristically, by evaluating danger value and possible landing points.

3) Spencer Schumann: A standard reactive heuristic controller from the example software augmented with a calculation of desired jump length based on internal simulation of Mario's movement. Incomplete tracking of enemy positions.

4) Mario Perez: Subsumption controller, inspired by controllers used in behaviour-based robotics.

5) Michal Tulacek: Finite state machine with four states: *walk-forward*, *walk-backward*, *jump* and *jump-hole*.

6) Rafel Oliveira: Reactive controller; did not submit any documentation.

7) Glenn Hartmann: Based on an example controller. Shoots continuously, jumps whenever needed.

D. Learning-based controllers

A number of controllers were submitted that were developed using learning algorithms in one way or another. These controllers exhibit a fascinating diversity; still, due to space considerations and paucity of submitted documentation, also these controllers will only be described summarily.

1) Matthew Erickson: Controller represented as expression tree, evolved with fairly standard crossover-heavy Genetic Programming, using *CompetitionScore* as fitness. The tree evaluates to four boolean values: left/right, jump, run and duck. Nonterminal nodes where chosen among standard arithmetic and conditional functions. The terminals were simple hard-coded feature detectors.

2) Douglas Hawkins: Based on a stack-based virtual machine, evolved with a genetic algorithm.

3) Alexandru Paler: An intriguing combination of imitation learning, based on data acquired from human playing, and path-finding. A* is used to find the route to the end of the screen; the intermediate positions form inputs to a neural network (trained on human playing) which returns the number and type of key presses necessary to get there.

4) Sergey Polikarpov: Based on the "Cyberneuron" architecture[8] and trained with a form of reinforcement learning. A number of action sequences are generated, and each is associated with a neuron; this neuron is penalized or rewarded depending on Mario's performance while the action sequence is being executed.

5) Erek Speed: Rule-based controller, evolved with a GA. Maps the whole observation space (22×22) onto the action space, resulting in a genome of more than 100 Mb.

VII. Results

The first phase of the competition was associated with the ICE-GIC conference in London, and the results of the competition presented at the conference. The results are presented in table I, and show that Robin Baumgarten's controller performed best, very closely followed by Peter Lawford's controller and closely followed by Andy Sloane et al.'s controller. We also include a simple evolved neural network controller and a very simple hard-coded heuristic controller (*ForwardJumpingAgent* which was included with the competition software and served as inspiration for some of the competitors) for comparison; none of the agents that were not based on A* outperformed the heuristic controller.

TABLE I

Results of the ICE-GIC phase of the competition.

Competitor	progress	ms/step
Robin Baumgarten	17264	5.62
Peter Lawford	17261	6.99
Andy Sloane	16219	15.19
Sergio Lopez	12439	0.04
Mario Perez	8952	0.03
Rafael Oliveira	8251	?
Michael Tulacek	6668	0.03
Erek Speed	2896	0.03
Glenn Hartmann	1170	0.06
Evolved neural net	7805	0.04
ForwardJumpingAgent	9361	0.0007

The second phase of the competition was associated with the CIG conference in Milan, Italy, and the results presented there. For this phase, we had changed the scoring procedure as detailed in section IV-A. A wise move, as both Robin Baumgarten's and Peter Lawford's agent managed to finish all of the levels, and Andy Sloane et al.'s came very close. In compliance with our own rules, Robin rather than Peter was declared the winner because of it being faster (having more in-game time left at the of all levels). It should be noted that Peter's controller was better at killing enemies, though.

The best controller that was not based on A*, that of Trond Ellingsen, scored less than half of the A* agents. The best agent developed using some form of learning or optimization, that of Matthew Erickson, was even further down the list. This suggests a massive victory of classic AI approaches over CI techniques. (At least as long as one does not care much about computation time; if score is divided by average time taken per time step, the extremely simple heuristic *ForwardJumpingAgent* wins the competition...)

[8]http://arxiv.org/abs/0907.0229

VIII. Discussion and analysis

While the objective for the competitors was to design or learn a controller that played Infinite Mario Bros as well as possible, the objective for the organizers was to organize a competition that accurately tested the efficacy of various controller representations and learning algorithms for controlling an agent in a platform game. Here we remark on what we have learned in each of these respects, using the software in your teaching and the future of the competition.

A. AI for platform games

By far the most surprising outcome of the competition was how well the A*-based controllers performed compared to all other controller architectures, including controllers based on learning algorithms. As A* is a commonly used algorithm for path-finding in many types of commercial computer games (e.g. RTS and MMORPG games), one could see this as a victory over "classical" AI over more fancy CI techniques which are rarely used in the games industry. However, one could also observe that the type of levels generated by the level generator are much less demanding and deceiving than those found in the real Super Mario Bros games and other similar games. All the levels could be cleared by constantly running right and jumping at the right moments, there were no hidden objects and passages, and in particular there were no dead ends that would require backtracking. All the A*-based agents consume considerably more processing time when in front of vertical walls, where most action sequences would not lead to the right end of the screen, suggesting that A* would break down when faced with a dead end.

While the sort of search in game-state space that the A* algorithm provides is likely to be an important component in any agent capable of playing arbitrary Mario levels, it will likely need to be complemented by some other mechanism for higher-level planning, and the architecture will probably benefit from tuning by e.g. evolutionary algorithms. Further, the playing style exhibited by the A*-based agents is nothing like that exhibited by a human player (the creepy exactness and absence of deliberation seems part of what made the *YouTube* video of Robin's agent so popular). How to create a controller that can (learn to) play a platform game in a human-like style is an industrially relevant (cf. *Demo Play*) problem which has not been addressed by this competition.

B. Competition organization

Looking at the objectives enumerated in section IV, we consider that the first two objectives (*ease of participation* and *transparency*) have been fulfilled, the third (*ease of finding a winner*) could have been met better and that we largely failed at meeting the fourth (*depth of challenge*).

Ease of participation was mainly achieved through having a simple web page, simple interfaces and letting all competition software be open source. Participation was greatly increased through the very successful media campaign, built on social media. Transparency was achieved through forcing all submissions to be open source and publishing them on

TABLE II

RESULTS OF THE ICE-GIC PHASE OF THE COMPETITION. EXPLANATION OF THE ACRONYMS IN THE "APROACH" COLUMN: RB: RULE-BASED, GP: GENETIC PROGRAMMING, NN: NEURAL NETWORK, SM: STATE MACHINE, LRS: LAYERED CONTROLLER, GA: GENETIC ALGORITHM.

Competitor	approach	progress	levels	time left	kills	mode
Robin Baumgarten	A*	46564.8	40	4878	373	76
Peter Lawford	A*	46564.8	40	4841	421	69
Andy Sloane	A*	44735.5	38	4822	294	67
Trond Ellingsen	RB	20599.2	11	5510	201	22
Sergio Lopez	RB	18240.3	11	5119	83	17
Spencer Schumann	RB	17010.5	8	6493	99	24
Matthew Erickson	GP	12676.3	7	6017	80	37
Douglas Hawkins	GP	12407.0	8	6190	90	32
Sergey Polikarpov	NN	12203.3	3	6303	67	38
Mario Perez	SM, Lrs	12060.2	4	4497	170	23
Alexandru Paler	NN, A*	7358.9	3	4401	69	43
Michael Tulacek	SM	6571.8	3	5965	52	14
Rafael Oliveira	RB	6314.2	1	6692	36	9
Glenn Hartmann	RB	1060.0	0	1134	8	71
Erek Speed	GA	out of memory				

the web site after the end of the competition. However, many competitors did not describe their agents in detail. In future competitions, the structure of such descriptions should be specified, and submissions that are not followed by satisfactory descriptions should be disqualified.

C. Using the Mario AI Competition in your own teaching

The Mario AI Competition web page, complete with the competition software, rules and all submitted controllers, will remain in place for the foreseeable future. We actively encourage use of the rules and software for your own events, and have noted that at least one local Mario AI Competition has already launched (at UC San Diego). Additionally, the software is used for class projects in a number of AI courses around the world. When organizing such events, it is worth remembering that the existing Google Group and its archive can serve as a useful technical resource, and that the result tables in this paper provide a useful point of reference. We appreciate if any such events link back to the original Mario AI Competition web page, and students are encouraged to submit their agents to the next iteration of the competition.

D. Future competitions

The 2010 Mario AI Championship, like the 2009 competition, uses a single website[9] but is divided into three tracks:

1) The gameplay track: Similarly to the 2009 competition, this track is aimed at producing the controller that gets furthest on a sequence of levels. However, the competition software is modified so that some of the levels are substantially harder than the hardest levels in last year's competition.

2) The learning track: This track is similar to the gameplay track, but favours controllers that perform online learning. Each controller will be tested a large number (e.g. 1000) of times on a single level, but only the score on the last attempt will count. The level will contain e.g. hidden blocks, shortcuts and dead ends. Thus, the scoring will reward controllers that learn the ins and outs of a particular level.

3) The level generation track: The level generation track differs substantially from the other tracks as what is tested is not controllers for the agent, but level generators that create new Mario levels that should be fun for particular players. The generated levels will be evaluated by letting a set of human game testers play them live at the competition event.

[9] http://www.marioai.org

REFERENCES

[1] J. Togelius, S. M. Lucas, H. Duc Thang, J. M. Garibaldi, T. Nakashima, C. H. Tan, I. Elhanany, S. Berant, P. Hingston, R. M. MacCallum, T. Haferlach, A. Gowrisankar, and P. Burrow, "The 2007 ieee cec simulated car racing competition," *Genetic Programming and Evolvable Machines*, 2008. [Online]. Available: http://dx.doi.org/10.1007/s10710-008-9063-0

[2] 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," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2008.

[3] J. Togelius, S. Karakaovskiy, J. Koutnik, and J. Schmidhuber, "Super mario evolution," in *Proceedings of IEEE Symposium on Computational Intelligence and Games (CIG)*, 2009.

[4] K. Compton and M. Mateas, "Procedural level design for platform games," in *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment International Conference (AIIDE)*, 2006.

[5] G. Smith, M. Treanor, J. Whitehead, and M. Mateas, "Rhythm-based level generation for 2d platformers," in *Proceedings of the International Conference on Foundations of Digital Games*, 2009.

[6] C. Pedersen, J. Togelius, and G. Yannakakis, "Modeling player experience in super mario bros," in *Proceedings of IEEE Symposium on Computational Intelligence and Games (CIG)*, 2009.

[7] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[8] I. Millington and J. Funge, *Artificial Intelligence for Games*. Morgan Kaufmann Pub, 2009.