The 2014 General Video Game Playing Competition

Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, Simon M. Lucas, *Senior Member, IEEE*, Adrien Couëtoux, Jerry Lee, Chong-U Lim, and Tommy Thompson

***Abstract*—This paper presents the framework, rules, games, controllers, and results of the first General Video Game Playing Competition, held at the IEEE Conference on Computational Intelligence and Games in 2014. The competition proposes the challenge of creating controllers for general video game play, where a single agent must be able to play many different games, some of them unknown to the participants at the time of submitting their entries. This test can be seen as an approximation of general artificial intelligence, as the amount of game-dependent heuristics needs to be severely limited. The games employed are stochastic real-time scenarios (where the time budget to provide the next action is measured in milliseconds) with different winning conditions, scoring mechanisms, sprite types, and available actions for the player. It is a responsibility of the agents to discover the mechanics of each game, the requirements to obtain a high score and the requisites to finally achieve victory. This paper describes all controllers submitted to the competition, with an in-depth description of four of them by their authors, including the winner and the runner-up entries of the contest. The paper also analyzes the performance of the different approaches submitted, and finally proposes future tracks for the competition.**

***Index Terms*—Competitions, evolutionary algorithms, general video game playing, Monte Carlo tree search, real-time games.**

I. Introduction

GAME-BASED AI competitions provide a popular way of benchmarking AI algorithms, and they have recently become a central feature of the AI/CI in games research field. In such competitions, AI methods are typically tested on their ability to play a game; competitors submit controllers which are evaluated in solo or competitive play. The main reason for testing algorithms through game-based competitions is that they provide fair and reliable benchmarks based on activities which are proven to be challenging for human cognition. But such competitions also have obvious appeal in that the results are easy to understand and interpret and they are, quite simply, fun. This contributes to raising awareness of AI research as well as to drawing competitors, and the software suites developed for competitions often double as student exercises in addition to their roles as "serious" AI benchmarks.

However, almost all existing competitions focus on a single game. This has the drawback that competitors overfit their solution to the specific game, limiting the value of the competition as a benchmark. If competitors are given the opportunity to tailor their solutions to the particular game, the results of the competition might reflect the ingeniousness of the competitor in engineering domain-specific adaptations, rather than the quality of the underlying AI algorithm. Therefore, it would seem that the promises of game-based AI benchmarking could only be fully realized if the competitor was not aware of which game(s) its submitted controller would get to play. One could also argue that the ability to play any of a number of unseen games is a valid approximation of artificial general intelligence, or "broad" AI, as opposed to "narrow" AI which is concerned with solving individual problems.

In this paper we describe one of the first competitions to focus not on the playing of a single game, but on a number of unseen games within a relatively large space of possible games. This competition focuses on video games, in particular the type of 2D arcade games that were common in the infancy of the video game medium, and is consequently called the General Video Game Playing Competition. The competition was run by the first five authors of this paper: D. Perez, S. Samothrakis, J. Togelius, T. Schaul, and S. M. Lucas; who also designed all the games and implemented the framework. The additional authors are those competition participants who scored highest in the contest, and those participants who also participated in a precompetition debugging phase.

The remainder of this paper is structured as follows. In the next section we highlight some of the most important related research concerning game description languages and game-based AI competitions. We then describe the general video game–AI (GVG–AI) framework in Section III, with particular focus on the video game description language (VGDL), followed by a description of the infrastructure and rules of the competition in Section IV. A long Section V contains descriptions of the controllers submitted to the competition: sample controllers developed by the organizers, and submitted controllers. Four submitted controllers are described in-depth by their developers (who also appear as authors of this paper). Finally, we describe

Manuscript received September 29, 2014; revised December 04, 2014 and January 28, 2015; accepted January 30, 2015. Date of publication February 10, 2015; date of current version September 13, 2016 . This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) under Grant EP/H048588/1, under the project entitled "UCT for Games and Beyond."

D. Perez-Liebana, S. Samothrakis, and S. M. Lucas are with the School of Computer Science and Electronic Engineering, University of Essex, Colchester CO4 3SQ, U.K. (e-mail: dperez@essex.ac.uk; ssamot@essex.ac.uk; sml@essex.ac.uk).

J. Togelius is with New York University, Brooklyn, NY 11201 USA (e-mail: julian@togelius.com).

T. Schaul was with New York University, New York, NY 10003 USA. He is now with Google DeepMind, London EC4A 3TW, U.K. (e-mail: schaul@gmail.com).

A. Couëtoux is with the Institute of Information Science (IIS), Academia Sinica, Taipei 115, Taiwan (e-mail: adrienc@iis.sinica.edu.tw).

J. Lee is with the Department of Computer Science and Information Engineering, National Taiwan Normal University, 106 Taipei, Taiwan (e-mail: jerry789520@gmail.com).

C.-U. Lim is with the Computer Science and Artificial Intelligence Laboratory (CSAIL), Massachusetts Institute of Technology, Cambridge, MA 02139 USA (e-mail: culim@mit.edu).

T. Thompson is with the University of Derby, Derby DE22 1GB, U.K. (e-mail: tommy@t2thompson.com).

Digital Object Identifier 10.1109/TCIAIG.2015.2402393

the results of the competition in Section VI. Final conclusions and guidelines for future work appear in Section VII.

II. Related Research

The idea of benchmarking AI through games is as old as AI itself. In particular, *Chess* was proposed as an AI benchmark by Turing, who proceeded to play the game using a manual execution of the Minimax algorithm, probably the world's first instance of AI game playing [1]. The World Computer Chess Championship series was inaugurated in the seventies as a way of allowing systematic comparison of AI approaches using this game, and has continued unabated for four decades [2]. These days, computers mainly compete with each other as the best computer programs have been able to beat the best humans since IBM's *Deep Blue* defeated the reigning human *Chess* champion, Garry Kasparov.

This victory of machine over man led to a reevaluation of the idea that you needed to be intelligent to play *Chess*, or indeed that *Chess*-playing ability was a good proxy of general intelligence at all, as the winning program essentially did tree search with plenty of domain-specific heuristics. But perhaps other games would be different? After all, playing *Chess* is different from playing *Go* and poles apart from playing *Halo* or *Grand Theft Auto*. Presumably, playing these games well requires very different cognitive skills.

In the last decade, a community of researchers has formed that focuses on applications of AI and CI in games, centred on the CIG and AIIDE conferences. In conjunction with these conferences, a number of competitions have been run, mostly organized as competition series with one or more competition events annually. These competitions have structured research activities in the field through the practice of competitors publishing their submissions as papers, and through the use of the software developed for the competitions as standard benchmarks.

The diversity of these competitions is considerable. There are competitions focused on board games, such as *Go* and *Othello*; first-person shooter games, in particular *Unreal Tournament 2004* [3]; platform games, such as *Super Mario Bros* [4] and *Geometry Friends* [5]; car racing, such as *TORCS* [6]; classic arcade games, such as *Ms Pac-Man* [7] and *X-Pilot* [8]; and real-time strategy games, such as *StarCraft* [9]. There are also competitions based on games tailor-made to test particular problem-solving capacities, such as the physical TSP game and competition [10], and several others that there is no space to discuss here. In addition to these conferences, the International Computer Games Association has hosted a Computer Olympiad since 1989, focusing on a number of board games and occasionally puzzles [11].

Though many of these competitions have been successful in the sense that they have had multiple entries that, over a succession of competition events, have advanced the state of the art in playing that particular game, they are still ultimately subject to the same limitations as the *Chess* championships in terms of benchmarking general intelligence. Many of the winning entries, especially in later iterations of any particular competition series, are based on domain-specific engineering solutions as much as general AI and machine learning algorithms. Success is often more about knowing the domain than developing good algorithms. This limits the amount we can learn from any particular competition. Although it is true that some techniques have arisen from research in particular games (such as transposition tables and iterative deepening from *Chess*, or Monte Carlo tree search from *Go*), an agent that wins one of these competitions would likely not perform above chance level in one of the others, even if it conformed to the same interface. Therefore, a competition designed with this in mind would allow us to learn more about artificial general intelligence.

One competition that has attempted to remedy this problem to some extent is the General Game Playing Competition (GGP), which ran for the first time in 2005 [12]. In this competition, agents are submitted and evaluated on their capacity to play a number of unseen games; the competitors do not know which games their agents will play when submitting. The games used in the GGP competition are usually variants of existing board games, or otherwise turn-based discrete games.

Another interesting initiative is the arcade learning environment (ALE), which is based on an emulation of the Atari 2600 game console [13]. The Atari 2600, released in 1977, was capable of implementing simple arcade games with rudimentary 2-D graphics. Nevertheless, some of these games were of very high quality and became classics, e.g., *Pac-Man*, *Adventure*, *Pitfall*, and *Frogger*. In ALE, the controller is presented with just the raw screen capture of the game, plus a score counter; in return, it outputs the set of buttons it chooses to press in each frame on the ALE's virtual Atari 2600 control stick.

Within the study of artificial general intelligence (AGI), there exists the idea that the intelligence of an agent can be quantified as its performance over a large number of environments. This rather simple idea has been formalized in several ways. In particular, it has been argued that an agent's intelligence can be estimated from its performance over a number of games sampled from a game space, and weighted by the lengths of their descriptions [14].

In order to be able to sample a space of games, or indeed to create a variety of new games, there needs to be a way of representing them; a game description language (GDL) is required (there also needs to be a game engine that can take a valid GDL description and execute it as a playable game). In the GGP competition, all games are expressed in a GDL based on first-order logic; syntactically, it's a version of Prolog [15]. The description language is rather low-level, and the description of a simple game such as *Tic-Tac-Toe* is several pages long. Further, this language is limited to deterministic turn-based finite games, and almost all of the games implemented in it are board games. Other authors have developed other GDLs, for other domains of games and/or at a higher level of abstraction (and, therefore, more compact). Such languages include the Ludi language for combinatorial board games [16] and the card game description language [17]. An overview of GDLs and other representations for game rules and mechanics can be found in [18].

The General Video Game Playing Competition is also based on a GDL, VGDL, which will be described in the next section. Unlike the language used for the GGP competition, VGDL is aimed at real-time games with a player avatar, potential stochastic effects and hidden information. The specification of games in a GDL allows an infinite a number of games, which distinguishes this competition from the ALE.

III. The GVG–AI Framework

A. A Video Game Description Language (VGDL)

The VGDL is designed around objects that interact in a 2-D space. All objects are physically located in a rectangular space (the screen), with associated coordinates (and possibly additional physical properties). Objects can move actively (through internally specified behaviors, or player actions), move passively (subject to physics), and interact with other objects through collisions. All interactions are local, subject to rules that depend only on the two colliding objects, and can lead to object disappearance, spawning, transformation, or changes in object properties. To a certain degree, nonlocal interactions can still be achieved, e.g., through projectile particles or teleportation effects. The avatar(s) are special types of objects, in that they are also affected by control signals from the bot (or human player).

A game is defined by two separate components, the (textual) level description, which essentially describes the initial positions of all objects and the layout of the game in 2-D, and the game description proper, which describes the dynamics and potential interactions of all the objects in the game. Each game terminates after a finite number of timesteps with integer score value together with an indication of whether the game was won or lost.

The game description itself is composed of four blocks of instructions. The *LevelMapping* describes how to translate the characters in the level description into (one or more) objects, to generate the initial game state. The *SpriteSet* defines the classes of objects used, organized in a tree, where a child class will inherit the properties of its ancestors. Further, class definitions can be augmented by keyword options. Objects can have a number of properties that determine how they are visualized (color, shape, orientation), how they are affected by physics (mass, momentum), and what individual behaviors they have (e.g., chasing, spawning others). One additional kind of property is resources, like health, mana, or ammunition, which increase or decrease as a result of interactions with other objects (finding health packs, shooting, taking fall damage). The *InteractionSet* defines the potential events that happen when two objects collide, e.g., bouncing, transforming, teleporting, replicating. Each such interaction maps two object classes to an event method. Finally, the *TerminationSet* defines different ways by which the game can end.

In order to permit the descriptions to be concise, an underlying ontology defines many high-level building blocks for games, including the types of physics used, movement dynamics of objects, and interaction effects upon object collisions.

Given these ingredients, VGDL can describe a wide variety of video games, including approximate versions of classic games like *Sokoban*, *Pong*, *Pac-Man*, *Lunar Lander*, *Dig-Dug*, *Legend of Zelda*, *Tank wars*, *Combat*, or *Mario Bros*.

For more detailed descriptions and possible uses of the language, we refer the interested reader to the report in which the initial ideas were laid out [19], and a follow-up paper [20] that fleshes them out, and includes a prototype written in Python (py-vgdl).

B. The GVG–AI Framework: Java–VGDL

Java–VGDL is the Java port of the initial implementation of VGDL (PY–VGDL), and is the framework used in the competition (also referred to here as the GVG–AI framework). The framework is able to load games and levels described in VGDL, exposing an interface that allows the implementation of controllers that must determine the actions of the player. The game engine also provides a forward model and information regarding the state of the game to the controllers. However, the VGDL definition of the game is not accessible to the controller (this is different to other competitions, such as GGP [12], [15]), so it is the responsibility of the agents to determine the nature of the game and the actions needed to achieve victory.

A controller in the GVG–AI framework must inherit from an abstract class and implement two methods. The first one is the constructor, which is called once per game and must finish before 1 s of CPU time.[1] The second method is the function `act`, called every game cycle, that must determine the next action of the agent in no more than 40 ms. If the agent takes between 40 and 50 ms to return an action then the NIL (no action) is applied. If the agent takes more than 50 ms to respond or if it exceeds the constructor budget time then it is disqualified from that particular game run.

Both methods receive a timer with information on when the call is due to end, and a `StateObservation` object that represents the current state of the game and provides the forward model. Thus, it is possible to advance the `StateObservation` object to the next state given an action, allowing the agent to analyze the effects of an action taken in the game, and to create copies of the `StateObservation` object. The games from the framework are generally stochastic in nature, and it is the responsibility of the agent to deal with this phenomenon.

The `StateObservation` object also provides the following information about the state of the game:

- Current time step, score and victory state (*win*, *loss*, or *ongoing*).
- List of available actions for the agent (or avatar). Each game in the framework can provide a different set of available actions for the player. This list can be a subset of all actions available in the framework: *up*, *down*, *left*, *right*, and *use*. The last action can trigger different effects depending on the game being played.
- List of observations. Each sprite in the game is given a type (identified with a unique integer) and a category (by apparent behavior): static, nonstatic, nonplayer characters (NPCs), collectables and portals (doors). All sprites in a game state are visible for the controller via observations, which contain information about their category, type and position. The `StateObservation` object provides a list of observations grouped by category (first) and sprite type (second).

[1]The competition was run in a dedicated server, Intel Core i5 machine, 2.90 GHz, and 4 GB of memory.

TABLE I
GAMES FEATURE COMPARATIVE. ALL GAMES OF THE COMPETITION ARE LISTED, DIVIDED INTO THE 3 GAME SETS: TRAINING (FIRST 10), VALIDATION (NEXT 10) AND TEST (LAST 10). LEGEND: I: INCREMENTAL; B: BINARY; D: DISCONTINUOUS; A0: ALL MOVES; A1: ONLY DIRECTIONAL; A2: LEFT, RIGHT AND USE. CHECK SECTION IV-B FOR A FULL DESCRIPTION OF THE MEANING OF ALL TERMS IN THIS TABLE

Game	**Score System**	**NPC Types**			**Resouces**	**Terminations**			**Action Set**
		Friendly	Enemies	> 1 type		Counters	Exit Door	Timeout	
Aliens	I		✓			✓			A2
Boulderdash	I		✓	✓	✓		✓		A0
Butterflies	I	✓				✓			A1
Chase	I	✓	✓			✓			A1
Frogs	B						✓		A1
Missile Command	I		✓			✓			A0
Portals	B						✓		A1
Sokoban	I					✓			A1
Survive Zombies	I	✓	✓		✓			✓	A1
Zelda	I		✓		✓		✓		A0
Camel Race	B		✓	✓			✓		A1
Digdug	I		✓		✓	✓			A0
Firestorms	B				✓		✓		A1
Infection	I	✓	✓			✓			A0
Firecaster	I				✓		✓		A0
Overload	I		✓		✓		✓		A0
Pacman	I		✓	✓	✓	✓			A1
Seaquest	D	✓	✓	✓		✓		✓	A0
Whackamole	I		✓					✓	A1
Eggomania	D		✓		✓	✓		✓	A2
Test Set Game 1	I		✓	✓	✓		✓		A0
Test Set Game 2	I		✓			✓		✓	A0
Test Set Game 3	I						✓		A1
Test Set Game 4	I	✓	✓			✓		✓	A0
Test Set Game 5	I		✓	✓		✓			A0
Test Set Game 6	D		✓	✓	✓	✓		✓	A0
Test Set Game 7	B						✓	✓	A1
Test Set Game 8	I		✓	✓	✓		✓		A0
Test Set Game 9	B						✓	✓	A1
Test Set Game 10	I	✓				✓		✓	A0

- Observation grid. The same group of observations provided to the player is also accessible in a grid format, where each observation is placed in a bidimensional array matching the positions of the sprites on the level.
- History of avatar events. An avatar event is a collision that happens between the avatar, or a sprite produced by the avatar, and any other sprite in the game. This list is sorted by game step, and each event provides information about the position where the collision happened and the type and category of each sprite in the event.

IV. THE GVG–AI COMPETITION

A. Server Infrastructure

There is a very simple server infrastructure, based on a combination of python/php scripts. The infrastructure is based on the concept of "instant gratification," i.e., that submitted agents/players should be evaluated as fast as possible. Each submitted agent is pushed into a queue, and dequeued one by one (so as to account for correct use of CPU time). Once at the head of the queue, the agent zip file is delegated to a python script which unzips the file, compiles it and runs it against a predefined set of games and levels. The script also performs basic error checking (e.g., propagates crashes or compilation errors) and attributes correct error levels to each type of problem. Once the run is over, the results are collected in a log file, which is then processed, beautified and pushed into a database. The information collected (e.g., score runs, possible failures) is presented to the users in a tidy html output.

B. Game Sets

The GVG–AI Competition featured 30 single-player games, each one of them with five different levels. These games are grouped into three different sets of 10 games each: training, validation and test. The training set is provided with the GVG–AI framework, and the description of the games can be seen in Table V.

The validation set was kept in the competition server, but its nature was unknown to the participants. During the competition, before the deadline, the contestants were able to submit their entries to the server, where the controllers were evaluated (in both the training and the validation sets) and the results posted on the competition website. Therefore, the participants were able to evaluate their controllers in unknown games and compare their performance in this set. After the competition, the games from this set were made public, and they are described in Table VI.

The third set (test) was employed to compute the final rankings of the competition. The participants were not able to know the identities of these games, and never got the chance to execute their controllers in this set. These games will be kept secret until the next edition of the competition, where they will be used as the validation set. Table I summarizes some of the features of all games (including the ones from the test set).

The *Score System* refers to the nature of the reward that the game provides via its score. In the binary (B) case, the only reward different than 0 is given when victory is achieved. For instance, in *Frogs* the score will be 1 only when the agent reaches the exit (note that the avatar must die during the game, however in *Frogs* this does not determine the score but the victory state). Other games have an incremental (I) score system, where different events provide a small increase in the score. An example of such a game is *Aliens*, where each enemy killed provides some additional score. Finally, the third category is discontinuous (D), which games use an incremental score system but include a particular event that provides a score gain a few orders of magnitude higher than the regular rewards obtained during the game. Obviously, those algorithms that are able to find this particular reward have a great advantage in these particular games. An example of this is *Eggomania*, in which killing the chicken produces 100 points, whereas saving each particular egg is only awarded 1 point. Note that the training set did not include any game of type D, with the aim of discovering if any of the algorithms submitted were able to discover this reward structure.

Another interesting aspect of these games is that not all of them provided the same set of actions to the player. The complete set is given in those games labeled as $A0$. $A1$ represents those games where the actions are only directional (there is no *use* action) and $A2$ indicates that the only available actions are *right*, *left*, and *use*.

Regarding NPC types, each game can be labeled as having *Friendly* (which do not harm the player) and/or *Enemies* (that cause damage to the player) NPCs. The column >1 type is ticked if there is more than one type of NPC (either friendly or enemy). The resources column indicates if the game contains sprites that the avatar can pick up or not.

Finally, different termination conditions for the game are established. The most general type is *Counters*, that refers to one or more type of sprite being created or destroyed. This includes cases such as when the avatar is killed (as, for instance, in *Seaquest*), or a certain number of sprites are captured by the player (as in *Butterflies*). Some games are finished when the avatar reaches a certain location or *Exit Door* (see *Firestorms*), and others end (either with a victory or a loss, depending on the game) when a certain timeout is reached, as in *Survive Zombies*. This termination is independent from the maximum number of game steps that can be played in a game, 2000, common for all games. This limit is introduced in order to avoid degenerate players never finishing the game, resulting in a loss if this limit is violated.

Table I shows that the 30 games of the GVG–AI framework differ significantly. The games were manually assigned to these three sets in order to obtain game sets whose features were distributed as evenly as possible. This way, there are no features in the final test set that have not been encountered before by the participants, either in the training or in the validation set.

All games present an assorted collection of winning conditions, different numbers of nonplayer characters (NPCs), scoring mechanics and even available actions for the agent. For instance, some games have a timer that finishes the game with a victory (as in *Whackamole*) or a defeat (as in *Sokoban*). In some cases, it is desirable to collide with certain moving entities (as in *Infection*, or in *Chase*) but, in other games, those events are what actually kill the player (as in *Portals*, or *Chase*).

These differences in game play make the creation of a simple game-dependent heuristic a considerably complex task, as the different mechanisms must be handled on a game per game basis. Furthermore, a completely domain-dependent controller would fail to successfully play the unseen games from the validation (during the competition) and test sets.

C. Evaluation

The entries to the competition were evaluated in the test set to obtain the final rankings. Each controller was executed ten times on each one of the five levels of each of the ten games, summing up to 500 games played in total.

For each particular game, three different statistics were calculated out of the 50 plays: number of games finished with a victory, total sum of points and total time spent. The controllers were ranked according to these three criteria, with the highest number of victories as the first objective to be considered. In the case of the same number of victories, controllers with the highest score were ranked first. Finally, the lowest time spent was used as the second tie-breaker.

Once the controllers were ranked on each game, points were awarded to them according to their position, following a Formula-1 score system. 25, 18, 15, 12, 10, 8, 6, 4, 2, and 1 points were given to entries ranked from the first to the tenth position, respectively. The rest of the entries obtained 0 points.

The winner of the competition was determined by the sum of points across all games. In the case of a tie, the controller with more first positions was considered better. If the draw persisted, then the number of second, third, etc. positions were taken into account, until the tie was broken.

V. Controllers

This section includes the description of the controllers submitted to the competition. First, detailed explanations of four agents are given by their developers. Then, the sample controllers are described, followed by a brief explanation of the other agents submitted to the competition. Each one of the entries is indicated with its final position in the rankings and the controller name.

A. 1st—Adrienctx (OLETS)—Adrien Couëtoux

1) Main Algorithm: Open-loop expectimax tree search (OLETS) is inspired by nierarchical open-loop optimistic planning (HOLOP, [21]). HOLOP samples sequences of actions and evaluates them by using the generative model of the problem, without actually storing states in memory. The sampled sequences are used to build a tree, whose nodes correspond to explored actions. This approach is nonoptimal for stochastic domains, but it saves considerable computing power, and actually performs well in practice [21]. OLETS has three big differences compared to HOLOP. First, it is designed for finite action spaces. Second, it does not use the average reward of simulations in a node to give it a score in the bandit formula, as is done in algorithms derived from upper confidence tree (UCT [22]). Instead, it uses our new method called open-loop expectimax (OLE). Last, OLETS does not use any roll-out and relies on the game scoring function to give a value to the leaves of the tree.

A node n stores the following data: $n_s(n)$ the number of simulations that have passed through n, $n_e(n)$ the number of simulations that have ended in n, $R_e(n)$ the cumulated reward from simulations that ended in n, $C(n)$ the set of children of n, and $P(n)$ the parent of n. $r_M(n)$ replaces the empirical average reward in the node scoring procedure and is the core of OLE. $r_M(n)$ is the weighted sum of two components. The first part is the empirical average reward obtained from simulations that exited in n. The second part is the maximum of n's children r_M values. See Algorithm 1 for more details.

Algorithm 1: OLETS

```
1: procedure OLETS($s, T$)
2:   $\mathcal{T} \leftarrow$ root                          ▷ initialize the tree
3:   while elapsed time $< T$ do
4:     RUNSIMULATION($\mathcal{T}, s$)
5:   return action $= \arg\max_{a \in C(root)} n_s(a)$
6: procedure RunSimulation($\mathcal{T}, s_0$)
7:   $s \leftarrow s_0$                                 ▷ set initial state
8:   $n \leftarrow$ root($\mathcal{T}$)                  ▷ start by pointing at the root
9:   $Exit \leftarrow$ False
10:  while $\neg Final(s) \wedge \neg Exit$ do           ▷ Navigating the tree
11:    if $n$ has unexplored actions then
12:      $a \leftarrow$ Random unexplored action
13:      $s \leftarrow$ ForwardModel(s, a)
14:      $n \leftarrow$ NewNode(a, Score(s))
15:      $Exit \leftarrow$ True
16:    else                       ▷ use node scoring to select a branch
17:      $a \leftarrow \arg\max_{a \in C(n)}$ OLE($n, a$)
18:      $n \leftarrow a$
19:      $s \leftarrow$ ForwardModel(s, a)
20:  $n_e(n) \leftarrow n_e(n) + 1$
21:  $R_e(n) \leftarrow R_e(n) + Score(s)$
22:  while $\neg P(n) = \emptyset$ do                    ▷ update the tree
23:    $n_s(n) \leftarrow n_s(n) + 1$
24:    $r_M(n) \leftarrow (R_e(n)/n_s(n)) + ((1 - n_e(n))/n_s(n))\max_{c \in C(n)} r_M(c)$
25:    $n \leftarrow P(n)$
26: procedure OLE($n, a$)
27:   return score $= r_M(a) + \sqrt{ln(n_s(n))/n_s(a)}$
```

2) Online and Offline Learning: We did not use any offline or online learning techniques in OLETS, because the computation time given at each step was very short. We decided to focus our efforts on the look ahead part of the algorithm. Still, adding offline learning techniques could greatly improve our controller, and should be a priority for future work, especially to handle more complex games.

3) Strengths and Weaknesses: The biggest weakness of OLETS is the absence of learning. As it stands, OLETS would do poorly on a difficult game that requires a long computation time to be solved. The main strength of OLETS is its computational simplicity. It is an open loop method, which means that it only stores sequences of actions. In comparison to closed loop methods that would store sequences of action-state, this reduces the search space dramatically, while inducing an additional error. Another strength is that OLETS uses OLE for its node scoring procedure. Because of this, high reward-paths are quickly identified, and information propagates to the root faster than when relying on traditional average reward.

The only game where OLETS obtained a very bad score was Camel Race. This poor performance is probably due to the fact that this game's rewards are only received many time steps after the right actions are chosen. Our algorithm does not use random walk simulations, and only sees the future as far as its tree branches go. This choice makes OLETS vulnerable on games where the rewards are very delayed.

4) Domain Knowledge: We added a taboo bias to the node scoring function. Our goal was to push the avatar to unexplored tiles of the screen when all other evaluation methods failed. Every time a node is added to the tree, if its state has been visited in the past T time steps, this node receives a taboo penalty of an order of magnitude smaller than 1. This showed good empirical results with $10 < T < 30$. We also added an ϵ-greedy exploration to the OLE procedure, with $\epsilon = 0.05$.

B. 2nd—JinJerry—Jerry Lee

1) Main Algorithm: JinJerry is inspired by Monte Carlo tree search (MCTS) [23] and uses an action selection strategy similar to the one introduced in [24]. It builds a one-level tree in every game cycle with the current game state as the root node and the one-action-ahead states of all actions as the leaf nodes. Then, it evaluates the immediate and potential score of all the leaf nodes. The immediate score is evaluated by the one-action-ahead state, and the potential score is evaluated by the state resulting from doing roll-outs of *SimulationDepth* extra random actions. A scoring heuristic evaluates states. The action selection strategy is to select the action with a safe state and a high score. If the random actions lead to a game-over state, the potential score is not considered. The potential score will replace the immediate score if it has a higher score. Algorithm 2 gives the pseudocode of JinJerry.

Algorithm 2: JinJerry

```
1: procedure JinJerry($s$)
2:   for each $a$ in $Actions$ do
3:     $s \leftarrow$ ForwardModel($s, a$)
4:     scores[$a$] $\leftarrow$ EvaluateState($s$)
5:     for $i \leftarrow 1$ to $SimulationDepth$ do
6:       $nextAction \leftarrow$ Random($Actions$)
7:       $s \leftarrow$ ForwardModel($s, nextAction$)
8:       if $Final(s)$ then
9:         break
10:    if $\neg Final(s)$ then
11:      $nextScore \leftarrow$ EvaluateState($s$)
12:      if $nextScore >$ scores[$a$] then
13:        scores[$a$] $\leftarrow nextScore$
14:  return $a \leftarrow \arg\max_a$ (scores)
```

2) Online and Offline Learning: The main design of JinJerry is on the scoring heuristic for evaluating a state. It considers the following factors:

- awayDist (chaseDist): is the distance between the agent and the closest "hostile" ("friendly") NPC;
- collectDist/movableDist/portalDist: is the distance between the agent and the closest resource/movable object/portal;
- visitedCount[x][y]: is the number of times a position (x, y) was visited by the agent;
- avatarResScore: is the total number of resources the agent collects;
- gameScore: is the actual game score;
- winScore: is set by the maximal (minimal) floating-point value when the agent wins (loses) the game.

Based on the above factors, we calculate five subscores

$$\begin{aligned}\text{NPC}_s &= (\text{awayDist} \times 3 - \text{chaseDist})/(X \times Y)\\ \text{Resource}_s &= 1 - \text{collectDist}/(X \times Y)\\ \text{Movable}_s &= \text{movableDist}/(X \times Y)\\ \text{Portal}_s &= 1 - \text{portalDist}/(X \times Y)\\ \text{Explore}_s &= 1 - \text{visitedCount}[x][y]/(X \times Y).\end{aligned}$$

In the above equations, X and Y denote the width and height of the game map. The product $(X \times Y)$ is used to make the distance-based scores smaller than one so that their effects are not greater than the actual game score, which is integral. The final score of a state is the weighted sum of the subscores, as shown in

$$\begin{aligned}&\text{Final}_s = \text{winScore} + \text{gameScore} + \text{avatarResScore} + 5\\ &\times\text{Explore}_s + 2 \times \text{Resource}_s + \text{Portal}_s + \text{NPC}_s + \text{Movable}_s.\end{aligned} \tag{1}$$

Normalizing the subscores into the same range [0, 1] also makes setting weights in the weighted summation easier. We started by setting all weights by one and examined the performance in the training set of games. Then, we gradually increased the weights of each subscore to see whether the increase is helpful. The pilot runs revealed that the exploration is very important for our controller to check unvisited areas and to avoid being stuck in a certain area. Getting resources is the main goal in some games and also worth a higher weight.

3) Strengths and Weaknesses: When the goal and structure of the game is simple and clear, *JinJerry* can lead the agent to the goal quickly. In games such as *Chase*, *Missile Command*, and *Survive Zombies*, it is relatively easy for the agent to achieve the goal. Thus, *JinJerry* performs well in these games and achieves high scores.

When the goal is complicated, as in the game *Sokoban* that asks the agent to move the boxes along nonblocking ways to suitable positions, *JinJerry* has no idea about how to achieve the goal. Another problem of *JinJerry* is that it uses Euclidean distance to estimate the distance between two positions. This causes *JinJerry* to get stuck on the map when there are many immovables (obstacles). In games such as *Zelda* and Boulderdash, getting stuck causes *JinJerry* to die easily. Furthermore, these games have "two-stage" goals—the agent needs to collect enough resources and then go to a portal. This is not considered in the design of *JinJerry*. *JinJerry* thinks of movable objects as bullets or hostile things and runs away. This makes it perform very badly, for instance, in the game *Eggomania*.

4) Domain Knowledge: Some domain knowledge is summarized from the training set of games and then is implemented in *JinJerry*. The principles include: 1) getting closer to the NPC, identifying whether it is hostile, and moving towards/away from it accordingly; 2) getting closer to resources; 3) getting closer to the portal when there is no resource; and 4) moving to unexplored areas.

C. 6th—Culim—Chong-U Lim

1) Main Algorithm: The controller uses a reinforcement learning technique called Q-learning [25] as an approach to general videogame playing. In Q-learning, the environment is modeled as a Markov decision process [26] with a set of states S and a set of actions A. For a given state $s \in S$, performing an action $a \in A$ results in a new state $s' \in S$, and an associated reward $r \in \mathbb{R}$ for this state-action pair. The aim of the learning is to approximate the *Quality* function $Q : S \times A \to \mathbb{R}$ using a Q-table (tabular Q-learning). This enables the controller to choose the best action for any given state, which provides the largest reward.

2) Online and Offline Learning: In order to compute the Q-table, the controller performs online learning—updating the Q-table during a game's run during each `act()` call using the following Q-table update loop.

1) Pick a random action $a \in A$ for given state s.
2) Performs the action a on s to obtain the next state s'.
3) Calculate the reward r for this state-action pair.
4) Updates the Q-table using the update function: $Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (r + \gamma \max_{a'} Q(s',a') - Q(s,a))$
5) Set $s \leftarrow s'$ and repeat.

Given the large state space, the resulting Q-table resembles a sparse matrix data structure. During the update, the learning rate α affects the controller's likelihood to override existing Q-table values. When $\alpha = 0$, no learning (i.e., updates from rewards) occurs and when $\alpha = 1$, the controller is always learning (i.e., updates using the most recent rewards.) The discount factor γ affects the prioritization of future rewards. With $\gamma = 1$, the controller will prioritize future rewards, while $\gamma = 0$ results in prioritizing only immediate rewards. We initialized our controller with $\alpha = 0.7$ and $\gamma = 1.0$. As each game run is nondeterministic, α was set to decay over time down to a minimum of $\alpha_{min} = 0.1$.

The controller also contains a depth parameter d—the maximum number of allowed update loop iterations (look-aheads). When the number of loop iterations exceeds d, learning restarts from the starting state s. This simulates having multiple instances of the controller performing learning on the same Q-table, but trading off having fewer look-aheads. We experimented and settled on $d = 10$ to balance sufficient look-aheads and multiple learning instances, while keeping within the time limit imposed on controllers for the competition.

While we experimented with offline learning by saving learned Q-tables, the final version of the controller did not include them. One reason was that the amount of time it took to read the large Q-tables was greater than the competition's allowed startup time. Also, each Q-table would have been specific to a given game and level, which might not have generalized well for the validation and test set games.

3) Strengths and Weaknesses: Q-learning is largely model-free and thus, the controller performs fairly well across all the games, particularly in the final test set used for rankings. However, it does not perform as well in games with large search spaces, especially with a large number of NPCs or movable objects, particularly those that can cause the player to lose (e.g., vehicles in *Frogger*.)

4) Domain Knowledge: The following sections describe areas in which domain knowledge of VGDL and its games were used.

a) ProofState meta-information: Due to the size of the `StateObservation` object in Java and the large size ($|S| \times |A|$) of the Q-table, storing them for Q-table lookups was not feasible. Hence, each Q-learning state object used the following meta-information for comparison:

- the position of the avatar;
- the average distance to all NPCs;
- the average distance to all portals;
- the grid position of the nearest NPC;
- the `id` of the nearest NPC;
- the number of NPCs remaining in the level.

While reducing the overall state space, the following points were observed. First, two `StateObservation` objects might differ in other aspects (e.g., absolute positions of each NPC,) but still be considered “equal.” Second, using real-values (e.g., squared-distances) results in a large range for comparison. However, with mostly grid-based games, we did not encounter significantly large variations. These tradeoffs did not factor heavily in our controller and made our approach feasible and performant for general video game playing.

b) Reward calculation: To calculate a given state-action pair's reward, we used a weighted sum of the following (signs in parenthesis denote contributing positively or negatively):

- the game's score $(+)$;
- the number of remaining NPCs $(-)$;
- the Manhattan distance to the closest NPC $(-)$.

If the game had ended, rather than using the game's score, a high positive (negative) reward was returned if the player had won (lost). In most games, fewer NPCs usually meant better progress towards winning (e.g., *Butterflies*, *Aliens*.) The Manhattan distance measure has been shown to be an effective general heuristic for progress in grid-based puzzle games [27].

D. 10th—T2Thompson—Tommy Thompson

1) Main Algorithm: This controller was largely inspired by controllers submitted to previous competitions that were reliant on simple, albeit effective methods. Notably, Robin Baumgarten's submission to the *Mario AI Competition* in 2009 which won the original gameplay track while reliant primarily on A* search [4]. Given the breadth of potential games that the VGDL permits, this controller was intended to measure how successful a controller reliant solely on state-based search could be.

The agent is reliant upon two methods: A* search and a steepest-ascent hill climber. In order for these to operate effectively, there is a need for heuristics. A collection of six heuristics were identified from the starter set that encapsulated the majority of gameplay objectives: shooting enemy sprites, killing enemy sprites with weapons, moving towards exit doors and collecting valuable artefacts. In addition, a basic win/lose heuristic and score improvement heuristic are provided as the default. These heuristics are managed at runtime and are selected based upon events occurring during play that increase the score. While the agent will start by searching to improve score, not die and potentially win, as it identifies what features of gameplay cause the score to increase it will become increasingly more focussed upon them.

The use of either A* search or hill climbing is dependent upon both the heuristics selected and the present state of the game. Each heuristic also identifies which search method to adopt while using it. The selection of method is based largely on experimentation by the author. However, in the event that the agent appears to be under threat, then the hill climber is adopted to find the safest escape route from the current state.

The A* implementation seeks to identify what the agent should do at a fixed step in the future. This is achieved by simulating the state of play 20 frames in the future. From this point, the A* then uses the time available to search for a path that maximises the selected heuristic(s).

2) Online and Offline Learning: The controller does not conduct any offline learning, given it is reliant upon the established set of hand-crafted heuristics mentioned previously. In order for these heuristics to be selected, online learning is adopted *per-se*. This is achieved using a fixed rule-set that assesses events that occur during play that result in an increase in score. If an event corresponds to a predefined rule, then a heuristic is selected for the agent to use for future action selection. For example, should the avatar fire a missile that eliminates a sprite and improves the overall score, then a rule will dictate the agent should now adopt the heuristic which is focussed on eliminating that sprite class using missiles.

3) Strengths and Weaknesses: A key strength of the agent is that once a heuristic that matches the gameplay has been identified, then often the agent can quickly maximize score. However, this is also one of the agent's largest weaknesses, given that it will conduct local search with a rather primitive heuristic until it can trigger an event that will subsequently help establish more intelligent search. In addition, if the rule set does not correspond an event with a heuristic, then the agent will not identify how to maximize score using this information.

An additional weakness of the agent is that any search conducted is reliant upon the forward model provided in the GVG–AI API. This may be stochastic and hence there is a probability that an A* search will be interrupted as a result of unpredicted future events.

4) Domain Knowledge: This agent is heavily reliant upon domain knowledge, given it adopts the aforementioned six heuristics, four of which were built by assessing raw data acquired from gameplay.

E. Sample Controllers

1) 3rd—Sample MCTS: This controller is an implementation of a vanilla MCTS algorithm. A full description of the algorithm, variants and applications can be found in [23]. Due to the real-time constraints of the framework, the iterations of the algorithm reach a final state very rarely. Therefore, the algorithm requires a function to evaluate states, which in this case is quite simple: in case a game is finished, a high positive (respectively, high negative) reward is given if the game was won (respectively, lost). Otherwise, the reward is the current score of the game.

The algorithm uses a play-out depth of 10 moves, an exploration-exploitation constant value of $\sqrt{2}$ and selects the most visited action at the root to pick the move to return for each game cycle.

2) 12th—Sample GA: The Sample GA controller is a rolling horizon open loop implementation of a minimalistic steady state genetic algorithm, known as a microbial GA [28]. At each timestep, the algorithm performs an evaluation between two different individuals from a sample population. Each individual is represented as an array of integers, with each integer denoting one possible action from the action space of a certain depth. By default the algorithm has a search depth of 7 (which means that each genome will have $7 * |A|$ genes, where $|A|$ is the number of actions). A tournament takes place between two individuals and the loser of the tournament is mutated randomly, with probability 1/7, whereas certain parts of its genome are recombined with parts from the winner's genome with probability 0.1. The procedure continues until the time budget is consumed. The evaluation function used is exactly the same as the one used by the Sample MCTS controller.

3) 14th—Random: This is the simplest controller provided within the framework. It just returns a random action at each game cycle, without any evaluation of the state to which this move takes the game.

4) 16th—One Step Look Ahead: This is a very simple controller that just moves the model one step ahead, evaluates the position using the same heuristic as Sample MCTS and selects the action with the highest evaluation score. The controller demonstrates simple use of the forward model.

F. Other Controllers

1) 4th—Shmokin: This is a hybrid controller, which uses A* as the default algorithm and switches to sample MCTS after a certain amount of steps of failing to find the goal.

2) 5th—Normal_MCTS: This entry is a variant of the Sample MCTS controller. The tree policy is changed adding a third term to the UCB1 formula [29]: the sum of rewards at that particular game state. Additionally, the action picked as the next move to make is the most visited one at the root, although this controller adds the sum of rewards as a tie breaker. The state evaluation function is the same as in the Sample MCTS controller.

3) 7th—MMBot: An MCTS controller that uses macro-actions of variable length (which cycles every 5 levels and ranges from 1 to 5) as part of the default policy. The heuristic used also tries to detect friend/foe NPCs.

4) 8th—TESTGAG: This controller is a variation of the Sample GA controller. The genetic algorithm is kept the same, but the state evaluation (or fitness) function is modified. As in the original controller, high positive or negative rewards are given for winning or losing the game, respectively. However, this controller provides a more involved score function in case the game is still ongoing, considering the distances to resources, NPCs and nonstatic sprites.

5) 9th—Yraid: This controller provides a more involved version of the sample GA controller, but with the addition that it incorporates "learning," i.e., it tries to identify which NPCS/items are harmful or helpful—that one can consider as some kind of online "training." It uses the information collected as part of the evaluation function.

6) 11th—MnMCTS: This entry is another controller based on Sample MCTS. The main differences consist of the use of discounted rewards during the play-outs and the use of a linear model to learn the importance of the features (such as distances to other sprites) in relation to the score change observed in the game. The predictions of this learnt model are used to bias the tree policy.

7) 13th—IdealStandard: This controller combines classical planning (i.e., tree search) with some form of online learning. It tries to guide the searches by associating types of items/NPCs with their properties (e.g., an NPC can kill you, thus better avoid it). Navigation through the level is performed by an A* path planning algorithm.

8) 15th—Tichau: This is the simplest controller received in the competition. It always, no matter the state or the game, or even the available actions, tries to apply the action *use*.

9) 17th—levis501: This is a stochastic, two-ply search controller. It performs a two-ply search and then proceeds with either playing the optimal move or chooses one move that does not make the controller lose the game at random. The two-ply tree search is performed using the same heuristic as sample MCTS and sample GA, while the randomness of actually play is increased every turn.

10) 18th—LCU_14: This controller does not use any Computational Intelligence technique. It basically consists of a rule based system that accounts for the presence of sprites close to the avatar. Action selection depends on the distances to these sprites, and also on the possession of resources that can be picked up. The controller incorporates domain knowledge about some specific games in the training set.

VI. Results

The participants of the competition were able to submit their controllers to both the training and the validation game sets before the deadline. Tables II and III show the final rankings in these two sets, respectively. It is interesting to analyze these results because participants could only see the games of the training set, while the ones from the validation set were hidden from them. Finally, Table IV shows the final results of the competition. The full rankings, including the results on each game, can be seen at the competition website (www.gvgai.net).

A first aspect to notice is that the first three entries are ranked in the same order in all sets, showing consistency with respect to

TABLE II
FINAL RANKINGS ON THE TRAINING SET. † DENOTES A SAMPLE CONTROLLER. G-1: ALIENS, G-2: BOULDERDASH, G-3: BUTTERFLIES, G-4: CHASE, G-5: FROGS, G-6: MISSILE COMMAND, G-7: PORTALS, G-8: SOKOBAN, G-9: SURVIVE ZOMBIES, G-10: ZELDA

Rank	Username	G-1	G-2	G-3	G-4	G-5	G-6	G-7	G-8	G-9	G-10	Total Points	Victories
1	adrienctx	**25**	**25**	6	18	10	12	**25**	**25**	18	**25**	189	40/50
2	JinJerry	18	10	18	**25**	**25**	**25**	15	4	**25**	8	173	33/50
3	SampleMCTS†	15	18	2	6	6	18	6	12	1	12	96	27/50
4	MnMCTS	2	12	8	15	2	10	12	10	12	6	89	25/50
5	MMbot	0	15	0	10	4	8	4	15	10	15	81	23/50
6	Shmokin	0	2	10	8	0	15	8	18	8	1	70	21/50
7	SampleGA†	8	0	**25**	4	8	0	0	8	0	10	63	19/50
8	Normal_MCTS	12	0	12	0	0	0	0	2	15	18	59	18/50
9	IdealStandard	0	0	4	12	0	4	10	0	0	0	30	13/50
10	LCU_14	4	1	15	0	1	0	0	0	6	0	27	13/50
11	Yraid	1	0	1	0	0	0	18	6	0	0	26	14/50
12	T2Thompson	0	4	0	1	15	2	0	0	0	4	26	15/50
13	levis501	0	0	0	0	18	0	0	2	0	0	20	10/50
14	OneStepLookAhead†	6	0	0	0	12	0	0	0	0	0	18	9/50
15	TESTGAG	10	0	0	2	0	1	2	0	0	2	17	15/50
16	culim	0	6	0	0	0	6	0	2	2	0	16	9/50
17	Random†	0	8	0	0	0	0	1	0	0	0	9	5/50
18	Tichau	0	0	0	0	0	0	0	0	4	0	4	7/50

TABLE III
FINAL RANKINGS ON THE VALIDATION SET. † DENOTES A SAMPLE CONTROLLER. G-1: CAMEL RACE, G-2: DIGDUG, G-3: FIRESTORMS, G-4: INFECTION, G-5: FIRECASTER, G-6: OVERLOAD, G-7: PACMAN, G-8: SEAQUEST, G-9: WHACKAMOLE, G-10: EGGOMANIA

Rank	Username	G-1	G-2	G-3	G-4	G-5	G-6	G-7	G-8	G-9	G-10	Total Points	Victories
1	adrienctx	0	**25**	**25**	18	**25**	**25**	**25**	**25**	**25**	10	203	24/50
2	JinJerry	10	18	15	**25**	15	15	12	10	4	1	125	14/50
3	SampleMCTS†	0	12	0	2	4	12	15	18	15	18	96	11/50
4	MnMCTS	4	1	8	8	12	6	6	12	12	**25**	94	12/50
5	Shmokin	6	8	0	12	10	10	18	6	18	6	94	13/50
6	levis501	8	6	18	0	18	0	1	0	0	12	63	8/50
7	culim	0	15	12	6	1	0	4	15	1	8	62	6/50
8	MMbot	0	10	0	10	0	8	0	8	8	15	59	9/50
9	IdealStandard	**25**	0	6	15	2	0	0	0	0	0	48	12/50
10	SampleGA†	0	0	0	0	10	18	0	0	0	0	28	6/50
11	T2Thompson	18	2	0	0	0	0	2	1	0	4	27	6/50
12	OneStepLookAhead†	15	0	10	0	0	0	0	0	0	0	25	6/50
13	Random†	0	0	0	4	10	0	8	0	0	0	22	4/50
14	Yraid	12	0	2	0	0	1	0	0	6	0	21	6/50
15	LCU_14	0	0	0	0	0	0	10	0	10	0	20	5/50
16	Normal_MCTS	2	4	0	0	0	4	0	4	2	0	16	6/50
17	TESTGAG	1	0	1	1	0	2	0	2	0	0	7	6/50
18	Tichau	0	0	4	0	0	0	0	0	0	2	6	1/50

TABLE IV
FINAL RESULTS OF THE GVGAI COMPETITION. † DENOTES A SAMPLE CONTROLLER

Rank	Username	G-1	G-2	G-3	G-4	G-5	G-6	G-7	G-8	G-9	G-10	Total Points	Victories
1	adrienctx	**25**	0	**25**	0	**25**	10	15	**25**	**25**	8	158	256/500
2	JinJerry	18	6	18	**25**	15	6	18	18	12	12	148	216/500
3	SampleMCTS†	10	18	6	4	18	**25**	6	12	0	0	99	158/500
4	Shmokin	6	**25**	0	12	10	8	0	10	6	0	77	127/500
5	Normal_MCTS	12	0	4	15	4	15	10	4	4	0	68	102/500
6	culim	2	12	8	1	8	4	8	6	10	2	61	124/500
7	MMbot	15	0	1	2	12	12	2	15	0	0	59	130/500
8	TESTGAG	0	8	15	0	0	1	1	0	2	**25**	52	68/500
9	Yraid	0	6	10	0	0	0	12	0	15	6	49	93/500
10	T2Thompson	0	0	0	10	0	0	0	1	18	18	47	87/500
11	MnMCTS	8	8	0	0	1	18	4	8	0	0	47	109/500
12	SampleGA†	4	10	12	0	0	2	0	0	8	4	40	76/500
13	IdealStandard	1	6	0	0	6	0	**25**	0	0	1	39	134/500
14	Random†	0	15	0	18	2	0	0	0	0	0	35	78/500
15	Tichau	0	6	0	8	0	0	0	0	1	15	30	55/500
16	OneStepLookAhead†	0	6	0	0	0	0	0	0	1	10	17	51/500
17	levis501	0	0	2	6	0	0	0	2	1	0	11	50/500
18	LCU_14	0	4	0	0	0	0	0	0	0	0	4	54/500

the quality of these controllers. The winner of the competition leads most of the games of the validation set, whereas its performance and that of the runner-up are very similar in training and test.

TABLE V
GAMES IN THE TRAINING SET OF THE GVGAI COMPETITION

Game	Description
Aliens	Similar to traditional Space Invaders, Aliens features the player (avatar) in the bottom of the screen, shooting upwards at aliens that approach Earth, who also shoot back at the avatar. The player loses if any alien touches it, and wins if all aliens are eliminated. 1 point is awarded for each alien or protective structure destroyed by the avatar and -1 point is given if the player is hit.
Boulderdash	The avatar must dig in a cave to find at least 10 diamonds, with the aid of a shovel, before exiting through a door. Some heavy rocks may fall while digging, killing the player if it is hit from above. There are enemies in the cave that might kill the player, but if two different enemies collide, a new diamond is spawned. 2 points are awarded for each diamond collected, and 1 point every time a new diamond is spawned. -1 point is given if the avatar is killed by a rock or an enemy.
Butterflies	The avatar must capture butterflies that move randomly around the level. If a butterfly touches a cocoon, more butterflies are spawned. The player wins if it collects all butterflies, but loses if all cocoons are opened. 2 points are awarded for each butterfly captured.
Chase	The avatar must chase and kill scared goats that flee from the player. If a goat finds another goat's corpse, it becomes angry and chases the player. The player wins if all scared goats are dead, but loses if it is hit by an angry goat. 1 point is awarded for killing a goat and -1 point for being hit by an angry goat.
Frogs	The avatar is a frog that must cross a road full of trucks and a river, only traversable by logs, to reach a goal. The player wins if the goal is reached, but loses if it is hit by a truck or falls into the water. 1 point for reaching the goal and -2 points for being hit by a truck.
Missile Command	The avatar must shoot at several missiles that fall from the sky, before they reach the cities they are directed towards. The player wins if it is able to save at least one city, and loses if all cities are hit. 2 points are given for destroying a missile, -1 point for each city hit.
Portals	The avatar must find the goal while avoiding lasers that kill him. There are many portals that teleport the player from one location to another. The player wins if the goal is reached, and loses if killed by a laser. 1 point is given for reaching the goal.
Sokoban	The avatar must push boxes so they fall into holes. The player wins if all boxes are made to disappear, and loses when the timer runs out. 1 point is given for each box pushed into a hole.
Survive Zombies	The avatar must stay alive while being attacked by spawned zombies. It may collect honey, dropped by bees, in order to avoid being killed by the zombies. The player wins if the timer runs out, and loses if hit by a zombie while having no honey (otherwise, the zombie dies). 1 point is given for collecting one piece of honey, and also for killing a zombie. -1 point if the avatar is killed, or it falls into the zombie spawn point.
Zelda	The avatar must find a key in a maze to open a door and exit. The player is also equipped with a sword to kill enemies existing in the maze. The player wins if it exits the maze, and loses if it is hit by an enemy. 2 points for killing an enemy, 1 for collecting the key, and another point for reaching the door with it. -1 point if the avatar is killed.

Another interesting analysis can be made of the percentage of victories achieved in the different sets. Many controllers are close to 50% of victories in the training set, with higher percentages achieved by the first two controllers in the rankings. However, in both validation and test sets, this performance drops significantly, with the winner of the competition being the only one to still achieve 50% of victories. Although this win rate is relatively good, it is far from optimal, and shows the difficulty of the problem tackled in this competition.

It is worthwhile highlighting that entries sorted by percentage of victories would rank differently. Controllers like *TESTGAG* in the training set, *MMBot* in validation, and *IdealStandard* in the test set would achieve a higher position if the rankings had been calculated this way. Nevertheless, the percentage of victories is not a good indication of the generality of the controllers, as this aggregates the victories in all games. The rankings per game system reward better controllers with a higher performance across many games.

In fact, the results of the competition show a clear gap between the two top controllers and the rest. The controllers that are placed in the first few positions demonstrate good general ability and are competitive in more than one game, achieving first to third positions on most games. The other controllers may excel in some games, but their performance is poor overall, ranking very low in most of them. This, of course, is reflected in the difference of points between these controllers.

An extreme case of this phenomenon is shown with the controller *IdealStandard*. This entry is mainly based on path planning techniques, with special emphasis on chasing resources and portals. Consequently, this controller achieves first position in the game G-1 (*Camel Race*) of the validation set. In this game, the objective is to reach the goal (a portal) quicker than other runners. It seems obvious that a path finding algorithm with high interest in reaching portals would perform very well, whereas other more general controllers, focused primarily on discovering features of the game, waste valuable time that makes them lose the race (it is especially interesting that the winner of the competition obtains 0 points in this game). Similar circumstances happen in the game G-7 of the test set, that shares some characteristics with *Camel Race*. This, however, does not mean that specialized controllers, like *IdealStandard*, are good in many other games. Obviously, path finding helps controllers to achieve a better performance, but more general approaches obtained better results.

Another point to analyze is how controllers that are variations of the provided sample ones perform, compared to these. On one hand, *TESTGAG* (a variation of *SampleGA*) does not perform better than its counterpart on the training and validation set, but surprisingly obtains better results on the test set. It seems that the modifications performed adapted the algorithm better to those final games. On the other hand, out of the many variations of *SampleMCTS*, none of them is able to achieve better results than the original controller, in any of the sets. It is not trivial to identify a reason for this, but a possibility is that adding domain knowledge to the algorithm does not help to achieve better performance in general (although the fact that performance is not good on the training set is still difficult to explain).

A final remark can be made about the *SampleMCTS* controller and the winner of the competition. *SampleMCTS* employs a

TABLE VI
GAMES IN THE VALIDATION SET OF THE GVGAI COMPETITION

Game	Description
Camel Race	The avatar must get to the finish line before any other camel. 1 point for reaching the finish line and -1 point if another camel reaches the finish line.
Digdug	The avatar must collect all gems and gold coins in the cave whilst digging its way through. There are enemies in the cave that kill the player on collision. Also, the player can at shoot boulders, which kill enemies, by pressing *use* two consecutive time steps. 1 point for every gem collected, 2 points for every enemy killed with a boulder and -1 point if the avatar is killed by an enemy.
Firestorms	The avatar must find its way to the exit while avoiding the flames spawned by portals from hell. The avatar can collect water as it goes. One unit of water saves the avatar from one hit of flame, but the game will be lost if flames touch the avatar and he has no water. -1 point each time a flame touches the avatar (whether he has water or not).
Infection	The avatar can get infected either through collision with bugs scattered around the level or with other infected animals (orange). The goal is to infect all healthy animals (green). Blue sprites are medics that cure infected animals and the avatar, but they can be killed by sword. 2 points for killing a doctor, 2 points for infecting a healthy animal and -1 point if the avatar is cured by the doctor.
Firecaster	The avatar must find its way to the exit by burning wooden boxes down. In order to be able to shoot, the avatar needs to collect ammunition (mana) scattered around the level. Flames spread, being able to destroy more than one box, but they can also hit the avatar. The avatar has health, that decreases when a flame touches it. If its health goes down to 0, the player loses. 1 point for each mana collected, 1 point for each box burnt and -2 points for each flame that touches the avatar.
Overload	The avatar must reach the exit with a determined number of coins, but if the amount of collected coins is higher than a (different) determined number, the avatar is too heavy to traverse the marsh and to finish the game. In this event, the avatar may kill marsh sprites with its sword, if it manages to collect one. 2 points for collecting the weapon, 1 point for clearing a marsh and 1 for each coin collected.
Pacman	The avatar must clear the maze by eating all pellets and power pills. There are ghosts that kill the player on collision if it hasn't eaten a power pill (otherwise, the avatar kills the ghost). There are also fruit pieces that must be collected. 40 points for killing a ghost, 10 points for eating a power pill, 5 points for eating a fruit piece, 1 point for eating a pellet and -1 point if a ghost kills the player.
Seaquest	The player controls a submarine that must avoid being killed by animals whilst rescuing divers by taking them to the surface. Also, the submarine must return to the surface regularly to collect more oxygen, or the avatar loses. The submarine's capacity is 4 divers, and it can shoot torpedoes at the animals. 1 point for killing an animal with a torpedo, and 1000 points for saving 4 divers in a single trip to the surface.
Whackamole	The avatar must collect moles that appear from holes. There is also a cat in the level doing the same. If the cat collides with the player, the game is lost. 1 point for catching a mole, -1 point for each mole captured by the cat and -5 points if the cat catches the avatar.
Eggomania	There is a chicken at the top of the level throwing down eggs. The avatar must move from left to right to avoid eggs breaking on the floor. Only when the avatar has collected enough eggs, can it shoot at the chicken to win the game. If a single egg is broken, the player loses the game. 1 point for each egg saved from crashing to the floor and 100 points for killing the chicken.

(nonoptimal) closed loop approach, storing the states of the game in the nodes of the tree without taking into account the stochasticity of the environment: during the *expansion* phase, the state of the new mode is reached by advancing the game with a specific action. The forward model provides a next state that depends only on a single advance call, resulting in a state that could be unrepresentative of the states reachable from that position. However, such a state will remain at that position in the tree during the rest of the iterations, producing a deceptive effect in the search.

An open loop planning algorithm, such as the competition winner, will accumulate the statistics based on the actions taken during the tree policy, resulting in a better approximate to the quality of an action. As can be seen, the difference between this controller and *SampleMCTS* in the test set is large in terms of points and percentage of victories. Both approaches are not optimal and can diverge, but an "incorrect" closed loop seems to be far worse. An interesting future approach is to create a "correct" close loop version where the forward model is sampled during tree policy as well.

VII. Conclusion

This paper described the rationale, setup and results of the first General Video Game AI competition. The competition attracted a reasonable number of entries and the leading two of these were able to significantly improve on the supplied Sample MCTS controller in terms of overall results.

Unlike the previous GGP competitions, this one is specifically tailored towards video games. Interestingly, despite the very different nature of the games involved in GVG versus GGP, MCTS has proven to be a leading approach in each type of competition, despite the fact that in GVG the 40 ms time limit places severe restrictions on the number of playouts that can be done for each move.

In the process of conceiving, planning and running the competition it has become clear that there are many ways to set up GVG competitions. The current one has already produced interesting results, and we now describe plans for future competitions both in terms of the main tracks and other variations.

- Gameplay track (current track): To date, competitors have only had a few months to develop their controllers, so we believe there is more to be learned by running this track for at least another year or two. The gameplay track provides a forward model and, while learning can be used to enhance performance, reasonable performance can already be obtained without any learning, as evidenced by the SampleMCTS controller finishing in third position.
- Learning track: Unlike the gameplay track, the learning track will not provide a forward model of the game which means that planning algorithms such as MCTS cannot be applied directly. This track will be set up to provide a testing ground for reinforcement learning algorithms. With no forward model a controller must learn while playing

the game which actions in which states tend to lead to higher rewards. To perform planning, an agent would have to learn an approximate model of the game while playing.

- PCG track: The aim of this track will be to find the best systems for generating game levels as well as entirely new games. We envisage using VGDL as the basis for this since it provides an elegant way to describe both game levels and game rules. This track will provide a new avenue for research in the area, building on previous work [16]–[18], [30]. More specifically, there has already been initial work on exploring the playability of randomly generated VGDL games [31].

We plan to run all the above tracks in the near future. Also under discussion is the concept of an oracle track. The aim of this track would be to investigate the extent to which the development of general game AI can be simplified by providing a full model (oracle). The oracle could be queried by setting the game engine to any state desired by the controller, and obtaining the score and transition outcomes (with associated probabilities) for any desired action. This puts the controller fully in charge of its own learning behavior. There would be a time constraint limiting the use of the oracle, so that managing this resource would be crucial, and may involve approaches such as active learning. However, it may be infeasible to provide this functionality for all GVG–AI games.

For each track there are also a number of orthogonal aspects of the games which can be varied to investigate particular areas of video game AI.

- Two player games: Currently all the games have been designed as single player games, for ease of initial setup and evaluation. However, two (or more generally N) player games are of obvious interest and should be catered for in the future. When running two player games care should be taken to properly account for player strength. Beating strong players should be worth more than beating weak players, especially in the event that the competition is flooded with a number of very similar weak players that a mediocre player has been designed to exploit. Also, the players may be heterogeneous, as in the Ghosts versus *Pac-Man* competition, for example, [7], [32].
- Games with continuous action spaces: Currently all our competition games have a limited discrete action space but in future it would be interesting to allow games with continuous action spaces such as *Lunar Lander* (though this could be approximated in the existing framework by integrating discrete actions over time).
- Games with partial observability: In the current set of games the StateObservation object passed to the agent contains the complete set of game objects, but the framework already supports passing a partial observation (such as the agent's immediate neighborhood, or objects within a particular field of view) of the game state, and this also offers an interesting dimension to explore.

The main initial aim of the competition was to provide a comprehensive evaluation service for general video game AI. This has already been achieved to a useful extent, but the value of this will grow significantly as we receive more entries and add more games.

It has also become increasingly clear that the framework may serve an equally useful role in providing an evaluation service for new video games. The idea here is that once we have a rich set of game AI controllers ranging in intelligence from stupid to smart we immediately have a powerful way to evaluate new games by measuring the different experiences of these bots when they play the game. The simplest measure is that smarter bots should get higher scores than stupid ones in any game of skill, but there will also be a rich set of other general statistics that can be applied and over time these can be correlated with human ratings of the same games. This has an obvious application in the PCG track, but in the long term may prove to be equally important for video game designers.

A further aim is to add value to all the tracks by enabling access to game replays via the web site, and also allowing human players to play all the publicly available games. The latter will be especially important for comparing the experiences of human players with bot players and analyzing the ability of various metrics to predict player experience.

Finally, all the current set of games have been very simply described in VGDL which is good for speed of development and ease of understanding, but the GVG–AI framework does not require this (the game agents see a standard interface that in no way depends on VGDL). We could also add non-VGDL games where the need arises, since VGDL is currently too restrictive to describe some of the subtler aspects of 2D video games which nonetheless significantly enhance the player experience.

Acknowledgment

The authors would like to thank all participants of the competition for their work and submitted controllers.

References

[1] A. M. Turing, “Chess,” in *Faster Than Thought*, B. V. Bowden, Ed. New York, NY, USA: Pitman, 1953, pp. 286–295.

[2] M. Newborn, *Computer Chess*. New York, NY, USA: Wiley, 2003.

[3] P. Hingston, “A new design for a Turing test for bots,” in *Proc. IEEE Conf. Comput. Intell. Games*, 2010, pp. 345–350.

[4] J. Togelius, N. Shaker, S. Karakovskiy, and G. N. Yannakakis, “The Mario AI Championship 2009–2012,” *AI Mag.*, vol. 34, no. 3, pp. 89–92, 2013.

[5] R. Prada, F. Melo, and J. Quiterio, Geometry Friends Competition 2014.

[6] D. Loiacono, P. L. Lanzi, and J. Togelius *et al.*, “The 2009 simulated car racing championship,” *IEEE Trans. Comput. Intell. AI in Games*, vol. 2, no. 2, pp. 131–147, 2010.

[7] P. Rohlfshagen and S. M. Lucas, “Ms Pac-Man Versus Ghost Team CEC 2011 Competition,” in *Proc. IEEE Congr. Evol. Comput.*, 2011, pp. 70–77.

[8] G. B. Parker and M. Parker, “Evolving parameters for Xpilot combat agents,” in *Proc. IEEE Symp. Comput. Intell. Games (CIG)*, 2007, pp. 238–243, .

[9] S. Ontanon *et al.*, “A survey of real-time strategy game AI research and competition in StarCraft,” *IEEE Trans. Comput. Intell. AI Games*, vol. 5, no. 4, pp. 293–311, 2013.

[10] D. Perez, P. Rohlfshagen, and S. M. Lucas, “The physical travelling salesman problem: WCCI 2012 Competition,” in *Proc. IEEE Congr. Evol. Comput.*, 2012, pp. 1–8.

[11] J. van den Herik, H. Iida, A. Plaat, and J. Hellemons, “The brain and mind-sports computer olympiad,” *ICGA J.*, vol. 36, p. 172, 2013.

[12] M. Genesereth, N. Love, and B. Pell, “General game playing: Overview of the AAAI competition,” *AI Mag.*, vol. 26, no. 2, p. 62, 2005.

[13] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, “The arcade learning environment: An evaluation platform for general agents,” *J. Artif. Intell. Res.*, vol. 47, no. 1, pp. 253–279, 2013.

[14] T. Schaul, J. Togelius, and J. Schmidhuber, “Measuring intelligence through games,” *CoRR*, vol. abs/1109.1314, pp. 1–19, 2011.

[15] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth, General game playing: Game description language specification 2008.

[16] C. Browne and F. Maire, "Evolutionary game design," *IEEE Trans. Comput. Intell. AI Games*, vol. 2, no. 1, pp. 1–16, 2010.

[17] J. M. Font, T. Mahlmann, D. Manrique, and J. Togelius, "A card game description language," in *Applications of Evolutionary Computing, EvoApplications 2013.*, ser. LNCS, A. I. Esparcia-Alcazar, Ed. *et al.* Vienna, Austria: Springer-Verlag, Apr. 3–5, 2013, vol. 7835, pp. 254–263.

[18] M. J. Nelson, J. Togelius, C. Browne, and M. Cook, "Rules and Mechanics," in *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, N. Shaker, J. Togelius, and M. J. Nelson, Eds. New York, NY, USA: Springer-Verlag, 2014, pp. 97–117.

[19] M. Ebner *et al.*, "Towards a Video Game Description Language," *Dagstuhl Follow-up*, vol. 6, pp. 85–100, 2013.

[20] T. Schaul, "A video game description language for model-based or interactive learning," in *Proc. IEEE Conf. Comput. Intell. Games*, 2013, pp. 193–200.

[21] A. Weinstein and M. L. Littman, "Bandit-based planning and learning in continuous-action Markov decision processes," in *Proc. 22nd Int. Conf. Autom. Plann. Sched.*, Brazil, 2012.

[22] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *ECML-06. Number 4212 in LNCS.* New York, NY, USA: Springer-Verlag, 2006, pp. 282–293.

[23] C. Browne *et al.*, "A survey of Monte Carlo tree search methods," *IEEE Trans. Comput. Intell. AI Games*, vol. 4:1, pp. 1–43, 2012.

[24] D. Robles and S. M. Lucas, "A simple tree search method for playing Ms. Pac-Man," in *Proc. 5th Int. Symp. Comput. Intell. Games*, 2009, ser. CIG'09, pp. 249–255.

[25] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, no. 3–4, pp. 279–292, 1992.

[26] R. Sutton, D. Precup, and S. Singh, "Between MDPs and Semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artif. Intell.*, vol. 112, pp. 181–211, 1999.

[27] C.-U. Lim and D. F. Harrell, "An approach to general videogame evaluation and automatic generation using a description language," in *Proc. IEEE Conf. Comput. Intell. Games*, 2014, pp. 1–8.

[28] I. Harvey, "The microbial genetic algorithm," in *Advances In Artificial Life. Darwin Meets von Neumann*. New York, NY, USA: Springer-Verlag, 2011, pp. 126–133.

[29] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Mach. Learn.*, vol. 47, no. 2, pp. 235–256, 2002.

[30] J. Togelius and J. Schmidhuber, "An experiment in automatic game design," in *Proc. IEEE Symp. Comput. Intell. Games*, 2008, pp. 111–118.

[31] G. Barros and J. Togelius, "Exploring a large space of small games," in *Proc. IEEE Conf. Comput. Intell. Games*, 2014, pp. 1–2.

[32] S. Samothrakis, D. Perez, P. Rohlfshagen, and S. Lucas, "Predicting dominance rankings for score-based games," *IEEE Trans. Comput. Intell. AI Games*, 2014, DOI: 10.1109/TCIAIG.2014.2346242.

Diego Perez-Liebana received the Ph.D. degree in computer science from the University of Essex, U.K.

He is now a Senior Research Officer with the University of Essex. He has published in the domain of Game AI, with research interests on reinforcement learning and evolutionary computation. He has organized several Game AI competitions, such as the Physical Travelling Salesman Problem and the General Video Game AI competitions, both held in IEEE conferences. He has programming experience in the video games industry with titles published for game consoles and PC.

Spyridon Samothrakis received the B.Sc. degree in computer science from the University of Sheffield, U.K., the M.Sc. degree in intelligent systems from the University of Sussex, U.K., and the Ph.D. degree in computer science from the University of Essex, U.K.

He is currently a Senior Research Officer at the University of Essex. His interests include game theory, machine learning, evolutionary algorithms, and consciousness.

Julian Togelius received the B.A. degree from Lund University, Sweden, in philosophy, the M.Sc. degree from the University of Sussex, U.K., in evolutionary and adaptive systems, and the Ph.D. degree from the University of Essex, U.K., in computer science.

He is an Associate Professor at New York University, New York, NY, USA, though most of this work was carried out while he was at the IT University of Copenhagen, Denmark. His prior interests were in the philosophy of mind, especially consciousness, but currently, he focuses on various applications of AI to games, including procedural content generation, player modeling, and general game AI.

Tom Schaul received the Ph.D. degree in computer science from the Technische Universität München, Munich, Germany, in 2011.

He was with New York University, New York, NY, USA, working as a Machine Learning Researcher. He is now with Google DeepMind, London, U.K., where he is interested in building general-purpose intelligent agents, and thinks that scalable game environments are a perfect testbed for that. His further interests include (modular/hierarchical) reinforcement learning, (stochastic/black-box) optimization with minimal hyperparameter tuning, and deep neural networks. He is also an author of the PyBrain open-source machine learning package.

Simon M. Lucas (M'98–SM'07) received the B.Sc. degree from Kent University, Canterbury, Kent, U.K., in 1986 and the Ph.D. degree from the University of Southampton, Southampton, U.K., in 1991.

He is a Professor of Computer Science at the University of Essex, U.K., where he leads the Game Intelligence Group. His main research interests are games, evolutionary computation, and machine learning, and he has published widely in these fields with over 180 peer-reviewed papers. He is the inventor of the scanning n-tuple classifier.

Prof. Lucas is the founding Editor-in-Chief of the IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES.

Adrien Couëtoux received the Ph.D. degree in computer science from the University Paris South (XI), France.

He is currently a Postdoctoral Fellow at Academia Sinica, Taipei, Taiwan. His research interests are reinforcement learning, optimization for power systems, and artificial intelligence in games.

Jerry Lee received the B.S. degree from the Department of Computer Science and Information Engineering (CSIE), National Taiwan Normal University (NTNU), Taiwan, in 2010.

He is currently working toward the M.S. degree. He is a member of the Metaheuristics Laboratory, CSIE, NTNU. His research interests include metaheuristics and related applications.

Chong-U Lim received the B.Eng. degree in computing from Imperial College London, U.K., in 2009 and the S.M. degree in electrical engineering and computer science from the Massachusetts Institute of Technology (MIT), Cambridge, MA, USA, in 2013.

He is currently pursuing the Ph.D. degree at MIT. His research interests span various topics of artificial intelligence and video games, such as player modeling, narrative generation, automated game design, and design evaluation.

Tommy Thompson received the B.Sc. degree from the University of Strathclyde in computer science, the M.Sc. degree from the University of Edinburgh in artificial intelligence, and the Ph.D. degree in computer science from the University of Strathclyde.

He is a lecturer of Computer Science at the University of Derby, U.K. His research interests include machine learning, evolutionary algorithms, automated planning and scheduling, and their applications within Game AI.