A proposal for combining reinforcement learning and behavior trees for regression testing over gameplay metrics

Pablo Gutiérrez-Sánchez[1 [0000−0002−6702−5726]],
Marco A. Gómez-Martín[2 [0000−0002−5186−1164]],
Pedro A. González-Calero[2 [0000−0002−9151−5573]], and
Pedro P. Gómez-Martín[2 [0000−0002−3855−7344]]

[1] PadaOne Games, Calle Profesor Jose Garcia Santesmases, 9, 28040 Madrid, Spain
pablo.gutierrez@padaonegames.com
[2] Complutense University of Madrid, Madrid, Spain

Abstract. In this paper, we propose a methodology based on reinforcement learning to automate video game testing. In particular, we discuss how the use of game playing agents with hybrid policies that incorporate reinforcement learning nodes alongside manually implemented flows in the context of behavior trees could lead to potentially more generalizable, understandable, and easier to fine-tune testing agents than those implemented via pure reinforcement learning techniques. In addition, we explore different strategies and configurations for training these agents, and describe a simple procedure for detecting significant modifications in selected gameplay metrics of a game level after introducing design changes. This is intended as a road map for future work in the development of automatic game testing and balancing tools.

Keywords: Automated game testing · Reinforcement learning (RL) · Behavior trees (BTs) · Game balance.

1 Introduction

Quality control in modern video games can be a major challenge. Nowadays it is not only necessary to keep a strict and continuous control of technical failures or bugs that may arise during the development process, but also of new problems in the playability derived from small changes in gameplay code and parameters that are used in different parts of the game. These changes can bring about various adverse effects such as preventing players from being able to complete previously solvable sections, altering the navigability of menus and environments, or modifying the difficulty perceived by the user, in dissonance with the experience originally conceived by the designers.

Quality assurance (QA) tasks usually involve an immense testing effort in which developers and players strive to detect and solve these problems. In light of this situation, in the last few years several research works have emerged with

the aim of proposing strategies to improve the QA process in video games and reduce its cost. Many of these methods are aimed primarily at developing agents (usually by using deep reinforcement learning, DRL) that act as synthetic players capable of automating checks that may otherwise require numerous hours of manual testing, in an attempt to redirect the efforts of human testers towards less mechanical and more creative tasks.

The methodology described in this paper can be applied to the design of automatic regression tests for detecting issues when modifications are made to level layouts, design parameters, or AIs that are reused in multiple sections of the game. The former may be performed during iterations of a specific level's design process, whereas the latter may be included when fine-tuning the behavior of a NPC in a level without thinking about the implications of those small variations on other levels where the NPC was placed before. Changes as simple as slightly modifying an enemy's movement speed could end up unexpectedly impacting how the player interacts with the game's levels. These changes need not be particularly drastic (such as as sudden violations of the completability of a level), and may boil down to the player taking more or less time to complete a part of the game, or exploiting a new way to beat a level that was not originally contemplated. All of these design alterations should not go unnoticed, but many of them are not usually straightforward to detect in typical testing environments.

In this paper we propose the application of different reinforcement learning methods to produce testing agents capable of interacting with a set of levels in a stealth game while collecting interaction statistics representative of the perceived gameplay in each level. The agents are afterwards used to test whether a change in the level or AI design induces significant changes in gameplay parameters collected by the agents before applying the modification.

2 Preliminaries

The following subsections introduce preliminary topics used in this work: RL, BTs, and RL-nodes in BTs.

2.1 Reinforcement Learning

Reinforcement learning (RL) is a process where an agent learns by trial and error from experience by interacting with its environment. In an RL problem, a distinction must be made between the agent, which interacts with its environment through actions, and the environment itself, which provides feedback and "rewards", or positive/negative reinforcement to the agent. RL algorithms usually take the formulation and formalism of Markov decision processes (MDPs) as a starting point. If we consider $s_t \in S$ as the state of the system at a given instant t, and $a_t \in A_{s_t}$ the action that the agent executes at that instant, where A_{s_t} is a possibly infinite set of admissible actions for the agent under state s_t, the agent's goal will be to learn a mapping between states and actions $\pi : S \to A = \cup_{s \in S} A_s$, that maximizes the expected long-run total reward from each state s:

$$G_t = E[\sum_{i=0}^{\infty} \gamma^i R_i] \tag{1}$$

where $\gamma \in (0, 1)$ is a discount factor that gives more weight to rewards earned in the short term, and R_i is the reward the agent receives at the i-th instant.

2.2 Behavior Trees

A Behavior Tree (BT) is a way of structuring policies or controllers in autonomous agents, such as robots or non-player characters in a video game (NPCs). In essence, a BT can be defined as a rooted tree in which the leaves correspond to execution nodes, associated to a specific task or condition check of the agent to be controlled, and the intermediate nodes take on the role of flow control nodes. BTs were initially conceived in the field of video game programming, with the first journal paper on BTs appearing in [6], and since then their use was progressively extended to the field of robotics. These constructs were originally developed with the motivation of promoting the production of modular, reusable, scalable, and easily understandable code when designing the behaviors of intelligent agents in the video game industry, thus trying to solve the problems commonly encountered with other classical mechanisms previously used, such as finite state machines (FSMs) or scripts.

The execution of a behavior tree starts at the root node, which generates signals or "ticks" with a given frequency. These signals enable the execution of a node and are propagated from parent to child in the internal hierarchy of the tree following the specific logic of each type of control node. A node can be executed if and only if it receives a tick. A child node can respond with *Running* to its parent if it is in the middle of an execution process, *Success* if it succeeds in completing its task, and *Failure* in any other case. Every result will eventually be propagated back to the root node, resulting in the generation of a new tick. This generates a cyclic behavior that continuously repeats the execution of the tree. This cyclic nature is precisely what gives BTs a good responsiveness to changes in the environment: since the conditions are rechecked at each iteration, the control flow can be modified if the context differs from that of the last cycle.

Control nodes represent composite behaviors that succeed or fail depending on the results returned from one or more of their children (which can be of any type, control or execution). The most common types of control nodes are *Sequence* and *Fallback*. *Sequence* nodes are used when a set of actions or conditions are intended to be executed sequentially, and will only succeed if all children complete their behaviors successfully. *Fallback* nodes on the other hand propagate the tick to their children from left to right until finding one that returns a *Success* or *Running* state, and then return that same state to their parent, meaning they will only return *Failure* when all of their children end with a failing state.

2.3 Reinforcement Learning Nodes in Behavior Trees

The above two concepts can be combined giving rise to behavior trees containing nodes supported by reinforcement learning. In general, the current literature contemplates two main types of RL nodes, depending on whether the learning is performed on a control or execution node.

- A *RL-control* node can be viewed as an extension of the usual control node, usually of type fallback [15], with the particularity that its children come to constitute the action space of a reinforcement learning algorithm. Thus, given a state s, the node reorders its children according to the expected rewards of executing each of them (being these estimates learned during training).
- A *RL-execution* node contains a *RL*-learning algorithm with states, actions and rewards defined by the designer. The idea here is to employ the hierarchical structure of a BT to learn relatively small actions that are only executed under controlled conditions.

In any of the above cases, the main goal of combining behavior trees with reinforcement learning is the implementation of actions and flows that are complex to program manually, but in specific regions of the agent's BT whose limited scope allows to dampen the so-called "curse of dimensionality" in learning subproblems. As an example, when designing the behavior of a robotic agent in an environment, it may be complicated to efficiently orchestrate actions that require the coordination of a large number of joints, such as "picking up an object". In this situation, it is possible to train an action node that learns only this task, and incorporate it into the original tree afterwards, with the problem being reduced to local learning in a controlled environment, but the behavior itself supported by the responsive structure of the BT.

3 Related Work

By testing we refer to the activity undertaken to evaluate the quality of a product and improve it by identifying its defects and problems [1]. Video games are complex software systems that must function correctly on different platforms with a range of configurations. The video game market is a very competitive one with buyers expecting increasingly more from them, which makes it unacceptable to release applications that are not robust or suffer from bugs. The robustness of a video game covers a wide spectrum of criteria, from the correct functioning of technical aspects such as performance or functional correctness to attributes such as the aesthetic soundness of the application. The validation of these criteria is a costly task in which a substantial part of the development effort of a project is invested, hence numerous strategies have been proposed in recent years in an attempt to automate these tasks or reduce their associated workload. One of the simplest alternatives is the use of game segments recorded manually by human testers, which are subsequently used to check that the replayed sequences

are still capable of completing the established objective [13]. However, when the structure or the game environment is modified, the tests generated by these methods are no longer valid, and it is necessary to once again resort to human testers to re-record new sequences for the modified scenarios. This continuous obsolescence naturally leads to the proposal of alternatives that are capable of adapting dynamically to variations in the game environments, giving rise to the use of AI-based agents for testing.

In [8], an AI played following the specification of a game given by a Petri net, making use of high-level actions, but required precise modeling of the level logic as well as manual implementation of the player's actions. DRL and IL techniques used in [16, 2, 3] show promising results, but most of these efforts are generally focused on detecting technical errors, or verifying whether an automated agent succeeds in completing given testing objectives within certain acceptable margins, with few references to the detection of subtle gameplay modifications that may undermine game design plans.

Moreover, these methods are not always trivial to implement, often requiring a potentially daunting process of trial and error in the choice of training algorithm, reward allocation policy, model features, or the hyperparameters of the underlying neural networks. Fortunately, over the last few years, libraries for popular game engines have been appearing that greatly facilitate this process, with ML-Agents [10] in the Unity 3D engine [18] being possibly one of the most well-known and actively maintained. Additionally, there has been work integrating reinforcement learning into hand-scripted control structures such as Behavior Trees (BTs) with the goal of narrowing learning problems to more controlled situations [15], thus reducing the time and effort needed to train agents.

In [9] the use of procedural personas for level playtesting characterized by different utility functions employing a variant of the Monte Carlo Tree Search (MCTS) is proposed. These enable the modeling of decision-making processes of players with different goals, play styles and personal preferences in simple scenarios such as the levels in the 2D dungeon crawler used for evaluation. However, the specification of the utility functions to be used is left to the designer, and the method's suitability for more complex environments requires further confirmation.

Lastly, there exist several works focused on facilitating the task of designing and adjusting the parameters of videogame contents, both in the field of playtesting and in level balancing and design. [7] proposes a method to obtain levels adjusted to a target difficulty based on the perception of an AI playing agent of maps generated by a trial-and-error algorithm. [17] describes a procedure for creating custom levels for platform games using preference learning based on surveys administered to players after completing each level, while [5] introduces a tool oriented to aid in the creation and understanding of procedural content generators with an emphasis on the analysis of their expressive range.

4 Methodology

In this paper we focus on the description of a preliminary automated testing methodology based on the agents presented in Section 2 for a specific game environment that will be covered in detail in Section 5.

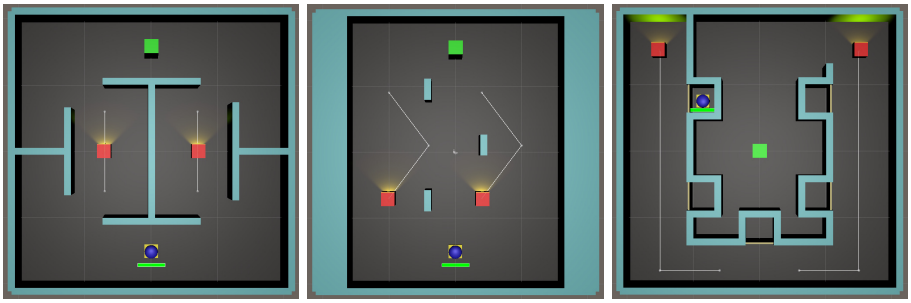
This methodology starts from a specific level of our game, which we consider finished from a design and/or development point of view. For this level, we wish to monitor certain gameplay parameters in order to detect any relevant changes induced by a modification on the enemy AI or the level design as soon as possible, for instance by tuning its parameters to meet the requirements of a later level. To carry out this tracking, we wish to train a set of "automatic players" using the techniques already described (RL, hybrid BTs) so that they learn to play the game "like humans". This learning process is conceived as a general procedure, which tries to generate agents capable of playing any kind of level, in such a way that the result may be reused for testing not only existing levels, but also new environments. This generality is, however, potentially challenging to achieve depending on the context, so the strategies proposed in this paper will need to be adequately validated in future experimental work.

Once agents with a sufficiently acceptable general ability have been obtained (where the concept of acceptability here takes on an admittedly subjective meaning), one can proceed to the generation of benchmark statistics on the parameters of interest at chosen stages after having agents playing at each level a large number of times. Once we have these references, it is possible to automatically perform a new round of gameplay and statistics compilation on the levels already developed to evaluate the possible impacts of the day's changes on their playability. With these statistics, we can verify if our agents are completing the levels with results equivalent to those recorded on the previous day. If this is statistically not the case, chances are that something has occurred, and we can issue a warning so that a member of the team can investigate the origin of this "alarm".

5 Practical Approach and Experiments

In this section, we will focus on concretizing the methodology described above to the proposal of automated tests for a simple demo game that includes some of the most common mechanics found in genres such as stealth games, in which the player navigates through levels patrolled by enemies that must be avoided while actively trying to attain a certain goal (often reaching a given point in the level). The rules of the designed game can be summarized in the following points:

1. The player is represented by a blue circular figure, with a continuous action space given by $A_t = \{(x, z), x, z \in [-1, 1]\}$, which determines the possible movement directions at each instant. The player has a fixed maximum velocity v^p, such that the velocity at each instant is given by $v_t^p = v^p \frac{(x_t, y_t)}{||(x_t, y_t)||}$.



(a) *Enemy Avoidance* level (b) *Enemy Timing* level (c) *Hiding Areas* level

Fig. 1. Example levels

2. The player starts each level placed on a yellow platform, to which he/she returns after being defeated by an enemy. The victory condition is to reach the green platform located at a fixed point in the level. A player is defeated when his/her health points (HP) are reduced to 0, these being lowered by 1 each time the player is hit by an enemy bullet and starting at 10 HP initially. Some sample levels can be found in Fig. 1.
3. The enemies are shaped like red cubes and have an associated "patrol" path (white lines in the figures), which they cycle through until they detect the player in their range of vision, after which they proceed to chase and shoot at their target as long as it is alive or remains within detection range.
4. Some levels contain yellow walls that block the enemies' range of vision, but have no effect on the player's movement or visibility (effectively creating hiding areas). Fig. 1c contains an example of this mechanic.

This demo game was developed in the Unity 3D engine, with the BTs of the enemies implemented using the Behavior Bricks plugin for the engine [14]. In the following subsections we describe the two training methodologies to be employed on the previously introduced demo game: first a method based on agents trained purely by reinforcement learning, and then a variant employing RL-execution nodes on hybrid behavior trees in an attempt to simplify the problem and provide a different family of policies.

5.1 Pure Reinforcement Learning Agents

We have already defined the action space A of our agent, but we still need to specify on the basis of what information these actions should be taken. A first approach could consider the following features:

– **Target location** (2 features). Components (x, z) of the distance vector between the player and the target platform, scaled to lay in the $[-1, 1]$ range.

– **Relative location of the enemies** (4 features per enemy). Components (x, z) of the distance vector between the player and each of the enemies in the level as well as the components (x, z) of the normal unit vector of the enemy's front face. The latter data provides information about the orientation of the enemy at any given time.

In addition to these features, we may also incorporate a series of spatial sensors or *raycasts* (for instance 12 sensors) that allow us to identify objects close to the agent in radial directions. Here we consider the objects "goal", "enemy" and "wall". The number of features derived from the incorporation of these sensors can be calculated as $(\text{number of rays} + 1) * (\text{number of detectable labels} + 2) = (12 + 1) * (3 + 2) = 65$ spatial features. This expression corresponds to a one-hot encoding of sensors and tags, where the $+2$ term in the number of detectable tags represents the cases where a ray detects an object with an unconsidered tag or nothing at all. It is interesting to note here that the length of the characteristics linked to the enemies can vary depending on the level, and this can be managed through the use of `BufferSensors` in ML Agents (which make use of an underlying attention module [19]).

Another alternative in this situation is to replace or complement the described observations by grid-based observations, taking advantage of the essentially two-dimensional nature of the game under consideration. These inputs, as the name suggests, are based on covering the level space by a grid of cells in which the presence of objects of the different categories to be perceived by the agent is detected, so that the observation associated with each cell is given by a one-hot representation of the objects found in it. This type of input is managed in ML Agents by means of `GridSensorComponents`, which provide what could be seen as a condensed and simplified version of a purely visual input. The total size of observations is thus given by GridSize.x∗GridSize.z∗number of detectable labels, and in the case of the levels considered for this game we can take $x = z = 20$, and keep the same tags as in the previous setup, but incorporating an additional one to represent the player. For the resulting agent to be reusable between levels, however, it is required to keep the dimensions of the grids fixed, which could hinder the general long-term viability of this method.

As for the reward allocation functions to employ, we can translate the ideas outlined in [9], replacing the notions of utility functions with specific reward functions for each type of agent we wish to produce. For instance, if we wish to create a bot for the example game with a conservative navigation style in which the agent tries to reach the goal without being spotted by enemies, we will need to implement a reward allocation function that penalizes enemy encounters and motivates effective use of hiding areas. One possibility for this last example may be given by the following scheme:

– Positive rewards: (+2 units) for reaching the goal and finishing the level.
– Negative rewards: (-2 units) for dying after being hit by an enemy bullet and finishing the level, (-0.25 units) for being hit by an enemy bullet, and (-0.1 units) every time the agent is detected by an enemy (at the instant it enters an enemy's vision range).

If, on the other hand, we were to create a bot that gives priority to reaching the goal in the shortest time possible, regardless of the number of life points at the end of the level, we could opt for the following configuration:

- Positive rewards: (+2 units) for reaching the goal and finishing the level.
- Negative rewards: (-2 units) for dying after being hit by an enemy bullet and finishing the level, (-0.1 units) for being hit by an enemy bullet, and an existential negative reward (-0.01 units) for each step taken by the agent.

This last penalty may be included to encourage agile flows and spur the agent to finish the episode as soon as possible to avoid too much negative reward.

In this situation, it is of particular importance to ensure that we do not reach a state of overfitting on the levels of the original battery of manually designed environments, trying in particular to avoid learning specific movement paths, or enemy dodging patterns conditioned by the structure of the starting levels. Justesen et al. [11] try to address this problem by using search-based procedural content generation (PCG), which can produce new learning levels to be used during the training loop of RL agents, eliminating to some extent the effect of overfitting on the produced bots. In our specific case, it is possible to use a randomized maze generation method, such as Prim's algorithm [4] to produce a basic level layout, subsequently eliminating certain random walls and incorporating enemies, hiding areas, and start and end points for the player.

5.2 Hybrid Behavior Tree - Reinforcement Learning Agents

In addition to agents trained purely by reinforcement learning, we consider a second class of agents given by a hybrid model between manually programmed behaviors and flows learned by RL in the context of a behavior tree. The idea here is to define a BT with a selector that alternates between fixed base flows (for instance "navigate to goal") and learned actions within RL nodes that are triggered upon entering certain states, e.g. when an enemy enters the agent's range of vision ("evade enemy"). In this way it is possible to set certain easily implementable actions, within level-agnostic action nodes and let the reinforcement learning algorithm handle specific situations, such as evading the enemies, under controlled conditions.

The reinforcement learning nodes in this case may consider the same features for the s_t states as those already described in the previous subsection, but it is also possible to opt for more limited subsets of these features, or even include new observations of novel types, depending on the particular needs of each action block. On the other hand, reward allocation features can be tailored locally to each RL node to motivate specific behaviors, and multiple functions that induce different play styles within the action can also be considered here.

This structure presents a number of possible expressive and practical advantages over the pure reinforcement learning based modality. Especially when the expected behavior of the game-playing agent is sufficiently complex, the division of the general flow into small blocks of localized logic allows on the one hand

to easily incorporate manually programmed sections to take a more rigid control over certain functional regions, and on the other hand to generate complex personalities, in the sense that the existence of local reward functions opens the possibility to create "action libraries" with blocks representing the same task to solve, but different styles to do so defined by the reward functions specified to train them. Thus, instead of defining a global personality for an agent, we can construct BTs that behave in different ways depending on the task at hand.

5.3 Testing Process

Once we have agents that are able to interact with the game in a satisfactory way, it is possible to proceed to the testing phase. At this stage we will be interested in extracting a sample of relevant execution parameters for a given number of attempts of the agent on the considered level. To do this, we may set a high number of repetitions per agent and level, in each of which we record statistics regarding, for instance, the HP with which the agent finishes each run and the number of steps needed to complete the level. These samples can be stored as references against which to test future changes in the level.

Having selected reference agents for a level and recorded a sample of contrast statistics, it is possible to perform a series of Student's t-tests (or Welch's t-tests if we do not assume variance equality) after incorporating any change in the game in order to check whether any of the statistics considered varies significantly (for the reference agent) after introducing the aforementioned change. To do this, it is sufficient to collect a new execution sample for the agent on the modified level and perform a parameter-by-parameter t-test to assess whether the means of the reference and test sets are significantly different from one another after the change ($\alpha = 0.05$). In our case, we assume that the variances of the two groups do not necessarily need to be equal, and therefore we propose the use of Welch's t-tests [12]. Under this scheme, we can evaluate the impact of a battery of design changes for each level and agent, from structural modifications such as adding or removing walls or hiding areas, to changes in the enemy AI such as updating its patrol speed, field of vision, gun firing rate, etc., all of which could lead to substantial design implications depending on the case. If normality is not assumed in the data, we can choose a non-parametric test such as Mann-Whitney, which only requires independence and that the variables to be compared are at least ordinal, to evaluate if the distributions of the two samples are the same.

6 Conclusions

This paper focused on the problem of creating testing agents to detect significant changes in different aspects of gameplay on a demo with stealth genre mechanics. We proposed and described two accessible alternatives to obtain agents capable of playing different levels of the game, the first one through pure RL, and the second through hybrid models that combine RL action nodes within hand-coded control structures in the form of BTs, both of them with the possibility

of introducing different reward function based play styles. Hybrid models, however, could potentially be easier to fine-tune and may possess greater expressive capacity than their pure RL counterparts. In this methodology, after obtaining satisfactory agents for each training method, a first simulation step can be carried out in which the agents play the considered levels repeatedly, recording relevant statistics regarding their perception of the execution, for instance HP remaining at the end of the level and time taken to complete it. Subsequently, after introducing various changes in the behavior of the enemies, the level layouts, and so on, new simulations can be performed with the agents on all levels to recollect performance metrics in the modified environments. Lastly, the reference samples may be compared with those of the post-change simulations by applying a means-contrast test or a non-parametric test on each of the parameters of interest to detect statistically significant changes in level gameplay.

This work is intended to serve as a road map and basis for future work aimed at analyzing whether the automated tests performed using this methodology can assist in locating and explaining changes of interest in the base of evaluated levels. Under the assumption that this is the case, if the tests after modification fail for any of the agents, the designer can analyze the results to check whether they conform to the predicted effect, if this was intended, or proceed to manually investigate the level where the test failed to search for the origin of the variation in the statistics. This would make it possible to narrow down the search for unexpected effects to those levels reported by the tests, or to evaluate whether a design change has the desired impact on the monitored parameters. Additionally, the introduction of game styles based on the use of different reward functions may allow to analyze the effect of applying changes not only at a general scale, but for different archetypal profiles that may be considered relevant in the design of the level, thus providing a tool to support the tailoring of experiences adapted to the player's profile.

References

1. Abran, A. (ed.): Guide to the software engineering body of knowledge, 2004 version: SWEBOK ; a project of the IEEE Computer Society Professional Practices Committee. IEEE Computer Society, Los Alamitos, Calif. (2004), oCLC: 934432015
2. Ariyurek, S., Betin-Can, A., Surer, E.: Automated Video Game Testing Using Synthetic and Humanlike Agents. IEEE Transactions on Games **13**(1), 50–67 (Mar 2021). https://doi.org/10.1109/TG.2019.2947597
3. Bergdahl, J., Gordillo, C., Tollmar, K., Gisslén, L.: Augmenting Automated Game Testing with Deep Reinforcement Learning. In: 2020 IEEE Conference on Games (CoG). pp. 600–603 (Aug 2020). https://doi.org/10.1109/CoG47356.2020.9231552, iSSN: 2325-4289
4. Buck, J., Carter, J.: Mazes for programmers: code your own twisty little passages. The Pragmatic Bookshelf, Dallas, Texas (2015), oCLC: ocn919295242
5. Cook, M., Gow, J., Smith, G., Colton, S.: Danesh: Interactive Tools For Understanding Procedural Content Generators. IEEE Transactions on Games pp. 1–1 (2021). https://doi.org/10.1109/TG.2021.3078323, https://ieeexplore.ieee.org/document/9426419/

6. Florez-Puga, G., Gomez-Martin, M., Gomez-Martin, P., Diaz-Agudo, B., Gonzalez-Calero, P.: Query-Enabled Behavior Trees. IEEE Transactions on Computational Intelligence and AI in Games **1**(4), 298–308 (Dec 2009). https://doi.org/10.1109/TCIAIG.2009.2036369, http://ieeexplore.ieee.org/document/5325892/
7. Gonzalez-Duque, M., Palm, R.B., Ha, D., Risi, S.: Finding Game Levels with the Right Difficulty in a Few Trials through Intelligent Trial-and-Error. In: 2020 IEEE Conference on Games (CoG). pp. 503–510. IEEE, Osaka, Japan (Aug 2020). https://doi.org/10.1109/CoG47356.2020.9231548, https://ieeexplore.ieee.org/document/9231548/
8. Hernández Bécares, J., Costero Valero, L., Gómez Martín, P.P.: An approach to automated videogame beta testing. Entertainment Computing **18**, 79–92 (Jan 2017). https://doi.org/10.1016/j.entcom.2016.08.002, https://www.sciencedirect.com/science/article/abs/pii/S1875952116300234
9. Holmgard, C., Green, M.C., Liapis, A., Togelius, J.: Automated Playtesting With Procedural Personas Through MCTS With Evolved Heuristics. IEEE Transactions on Games **11**(4), 352–362 (Dec 2019). https://doi.org/10.1109/TG.2018.2808198, https://ieeexplore.ieee.org/document/8295256/
10. Juliani, A., Berges, V.P., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., Gao, Y., Henry, H., Mattar, M., Lange, D.: Unity: A General Platform for Intelligent Agents. arXiv:1809.02627 [cs, stat] (May 2020), http://arxiv.org/abs/1809.02627, arXiv: 1809.02627
11. Justesen, N., Torrado, R.R., Bontrager, P., Khalifa, A., Togelius, J., Risi, S.: Illuminating Generalization in Deep Reinforcement Learning through Procedural Level Generation. arXiv:1806.10729 [cs, stat] (Nov 2018), http://arxiv.org/abs/1806.10729, arXiv: 1806.10729
12. Lu, Z., Yuan, K.H.: Welch's t test, pp. 1620–1623 (01 2010), 10.13140/RG.2.1.3057.9607
13. Ostrowski, M., Aroudj, S.: Automated Regression Testing within Video Game Development. GSTF Journal on Computing (JoC) **3**(2), 10 (Aug 2013). https://doi.org/10.7603/s40601-013-0010-4, http://www.globalsciencejournals.com/article/10.7603/s40601-013-0010-4
14. PadaOne Games: BehaviorBricks. http://bb.padaonegames.com/ (2021), accessed: 2021-04-16
15. Pereira, R.d.P., Engel, P.M.: A Framework for Constrained and Adaptive Behavior-Based Agents. arXiv:1506.02312 [cs] (Jun 2015), http://arxiv.org/abs/1506.02312, arXiv: 1506.02312
16. Pfau, J., Smeddinck, J.D., Malaka, R.: Automated Game Testing with ICARUS: Intelligent Completion of Adventure Riddles via Unsupervised Solving. In: Extended Abstracts Publication of the Annual Symposium on Computer-Human Interaction in Play. pp. 153–164. ACM, Amsterdam The Netherlands (Oct 2017). https://doi.org/10.1145/3130859.3131439, https://dl.acm.org/doi/10.1145/3130859.3131439
17. Shaker, N., Yannakakis, G., Togelius, J.: Towards automatic personalized content generation for platform games (12 2010)
18. Unity Technologies: Unity Real-Time Development Platform | 3D, 2D VR & AR Engine. https://unity.com/ (2021), accessed: 2021-04-10
19. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention Is All You Need. arXiv:1706.03762 [cs] (Dec 2017), http://arxiv.org/abs/1706.03762, arXiv: 1706.03762