# Go-Explore Complex 3D Game Environments for Automated Reachability Testing

Cong Lu[1,*], Raluca Georgescu[2,†] and Johan Verwey[2,†]

[1]University of Oxford
[2]Microsoft Research

Abstract
Modern AAA video games feature huge game levels and maps which are increasingly hard for level testers to cover exhaustively. As a result, games often ship with catastrophic bugs such as the player falling through the floor or being stuck in walls. We propose an approach specifically targeted at reachability bugs in simulated 3D environments based on the powerful exploration algorithm, Go-Explore, which saves unique checkpoints across the map and then identifies promising ones to explore from. We show that when coupled with simple heuristics derived from the game's navigation mesh, Go-Explore finds challenging bugs and comprehensively explores complex environments *without the need for human demonstration or knowledge of the game dynamics.* Go-Explore vastly outperforms more complicated baselines including reinforcement learning with intrinsic curiosity in both covering the navigation mesh and number of unique positions across the map discovered. Finally, due to our use of parallel agents, our algorithm can fully cover a vast 1.5km x 1.5km game world within 10 hours on a single machine making it extremely promising for continuous testing suites.

**Keywords**
Automated Game Testing, Go-Explore, Reinforcement Learning

## 1. Introduction

With the scope and size of modern AAA games ever increasing, QA teams are struggling to exhaustively test these games [1, 2]. It is becoming increasingly common for games to ship with many bugs that are discovered by end-users and only get fixed several months following the title's release [3, 4]. The challenge of covering every corner of a multi-square mile game world is exacerbated by the fact that the maps constantly change during development which requires tests to be repeated hundreds of times.

A large proportion of bugs that occur in game worlds are related to reachability [5]. Game testers need to verify that every area of a map that should be reachable through the player's abilities are reachable and haven't been inadvertently blocked off during iteration. Some areas are designed to test the player's skill and are hard to reach, but not impossible. Finally, there are those areas of the map that should not be reachable—in these areas, players could fall through the floor or find themselves inside geometry that is not part of the gameplay area. Automating reachability testing for large game maps would alleviate a huge burden during game development and allow bugs to be identified earlier without human labor.

A large body of prior work has considered ways to automate these tests including by imitating recorded human actions [6, 7]. However, recorded actions are not robust to level changes, rely on environment determinism, and cannot verify that forbidden areas are unreachable. Other approaches include those that combine reinforcement learning with curiosity-based reward bonuses [8, 9, 10]. Even so, these methods must use approximations for the vast observation spaces that feature in modern games, and have been shown under these settings to be brittle to catastrophic forgetting [11].

In contrast, in this paper, we investigate Go-Explore [11], a simple but powerful exploration algorithm which maintains a cache of discovered locations, and identifies promising locations with heuristics to reset and explore from. Whilst Ecoffet et al. [11] observed that Go-Explore was able to discover a known reachability bug—the "treasure room curse" [12]—in the course of solving Montezuma's revenge; to the best of our knowledge, ours is the first application to 3D environments. We show that Go-Explore can readily handle parallelization and rapidly explore vast game worlds of many square miles. Furthermore, we find empirically that random exploration is asymptotically better than learned exploration such as Random Network Distillation [13] under the same conditions which makes our approach easier to implement.

The core contributions of this paper are as follows:

1. We apply a simple and efficient algorithm, Go-Explore, for discovering reachable positions in hard exploration maps without human demonstrations or knowledge of the game dynamics.

*The Joint Workshop Proceedings of the 2022 Conference on Artificial Intelligence and Interactive Digital Entertainment*
*Corresponding author.
†These authors contributed equally.
cong.lu@stats.ox.ac.uk (C. Lu); raluca.georgescu@microsoft.com (R. Georgescu); joverwey@microsoft.com (J. Verwey)

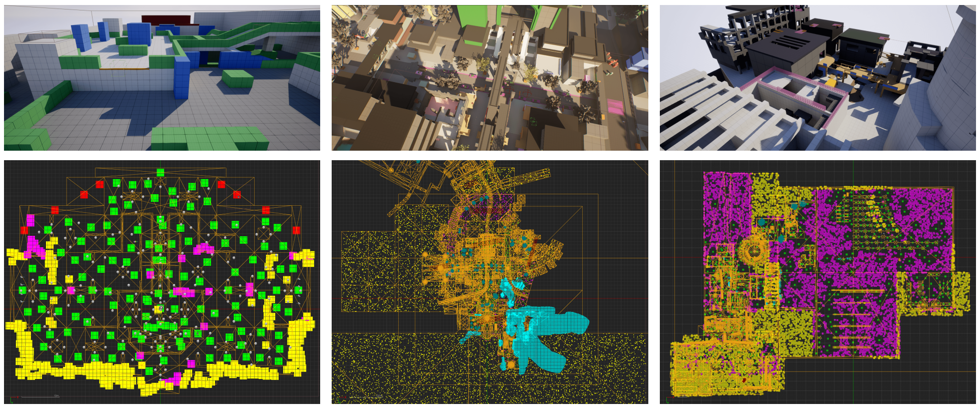
(a) Small Map (b) Large Map (c) Traversal Map

**Figure 1:** We evaluate Go-Explore for reachability testing on three proprietary test game levels of increasing difficulty designed in Unreal Engine 5. The smallest is 100m x 100m and the largest 1.5km x 1.5km which matches the scale of modern game maps. We initially sample points from the nav-mesh which act as 'goals' for exploration. Reached and unreached goals at the end of training are colored in **green** and **red** respectively. We additionally color non-goal points we discover as **magenta** if they are within 3m of the nav-mesh and yellow otherwise. These yellow points are discovered positions that an agent would not be able to reach by simply following the nav-mesh, and are often indicative of an unexpected bug. For example, on the small map, the yellow points on the south wall point to an incorrectly configured section which allows the agent to reach the top of the wall and fall off the map leading to undefined behavior. **Yellow** points on the left and bottom of the large map show places where the agent can fall off the map.

2. We propose a criteria to classify discovered points into expected and unexpected by comparison to the game navigation mesh. Undiscovered regions of the navigation mesh may also be flagged.
3. We show that by parallelizing agents within a single game instance, we can exhaustively cover vast game maps of the size 1.5km x 1.5km within 10 hours on a single machine, making our algorithm extremely promising for continuous testing suites.

## 2. Preliminaries

We begin by introducing traditional pathing in video games via navigation meshes, the reinforcement learning (RL) paradigm, and exploration algorithms in RL. In this paper, we consider the problem of automatically testing for "reachability" which we define to be determining the set of locations of the map an agent can reach. This is distinct from the traditional definition of graph reachability as we assume no knowledge of the environment dynamics to connect adjacent positions, and actions may have stochasticity.

### 2.1. Navigation Meshes.

AI agents in modern video games typically use a navigation mesh [14, 15] or "nav-mesh" to navigate from one area of a map to another. A nav-mesh is a collection of two-dimensional convex polygons [16] which define areas of the map that are traversable by agents. Within each polygon, an agent can freely move without being obstructed by any environment obstacles such as trees and rocks. Adjacent polygons are connected together in a graph and pathfinding between them can be done with classic graph search algorithms such as A* [17].

A naïve method of verifying reachability may be to send an agent to every node in the nav-mesh, one by one. However, the nav-mesh may not cover the entire map and there are often gaps between patches of the nav-mesh that cannot be traversed by just walking. Instead, we may use the nav-mesh as a starting point for positions we expect our agent to reach. By comparing the results of an exploration algorithm vs. this ground truth, we can *categorize observed positions as expected or unexpected* and identify positions on the nav-mesh that we expect to reach but have not been reached which we illustrate in Figure 1.

### 2.2. Reinforcement Learning.

A natural way to accelerate game testing would be to try and train agents to explore and find novel states. Reinforcement Learning [18, 19] is a successful paradigm for learning intelligent agents where optimal behavior may be specified by a reward function. We model the game environment as a Markov Decision Process (MDP, Sutton and Barto [19]), defined as a tuple $M = (\mathcal{S}, \mathcal{A}, P, R, \rho_0, \gamma)$, where $\mathcal{S}$ and $\mathcal{A}$ denote the state space and action space respectively, $P(s'|s,a)$ the transition dynamics, $R(s,a)$ the reward function, $\rho_0$ the initial state distribution, and $\gamma \in (0,1)$ the discount factor. The goal in RL is to optimize a policy $\pi(a|s)$ that maximizes the expected discounted return $\mathbb{E}_{\pi,P,\rho_0}\left[\sum_{t=0}^{\infty}\gamma^t R(s_t,a_t)\right]$.

## 2.3. Exploration in RL.

Count-based exploration describes a family of algorithms that assign reward bonuses based on visitation count to encourage agents to seek out novel states. For finite state MDPs, we define the visitation count $n(s)$ of a state $s \in S$ to be the number of times a particular state has been encountered. Prior methods [20, 21] have proposed to assign a reward bonus of $r_t = 1/n(s)$ or $r_t = 1/\sqrt{n(s)}$ at each time step. These methods may be extended to MDPs with continuous state spaces by treating $n(s)$ as a density. Even so, count-based methods are hard to scale to larger state spaces and cannot naturally handle environments in parallel.

Random Network Distillation (RND, Burda et al. [13]) is a flexible way to compute an exploration bonus for high-dimensional state spaces by estimating the error of a neural network predicting features of the game observations given by a fixed randomly initialized neural network. Concretely, the fixed randomly initialized *target* network $f : S \to \mathbb{R}^k$ computes an embedding and the *predictor* network $\hat{f}_\theta : S \to \mathbb{R}^k$ is trained by gradient descent to minimize the prediction error $||f(s) - \hat{f}_\theta(s)||^2$ with respect to its parameters, $\theta$. This prediction error is then used as the reward bonus. Over the course of training the randomly initialized neural network $f$ is distilled into $\hat{f}_\theta$. Intuitively, the prediction error is expected to be higher for novel states dissimilar to the ones that the predictor has been trained on.

## 2.4. Go-Explore.

Go-Explore [11] is an alternate approach for hard-exploration problems which maintains a cache of previously explored states and then uses heuristics to periodically select promising states and then explore from them. Once sufficiently high return trajectories are found (first phase, exploration), the discovered behavior is made robust with imitation learning [22] (second phase, robustify). This algorithm aims to avoid the phenomenon of "derailment" that may occur in algorithms which do not have explicit memory such as RND where the predictor network may lose information about previously reached states due to catastrophic forgetting. This phenomenon could be even likelier to occur in large maps.

The original Go-Explore [11] assumes full access to a simulator of the environment and the ability to reset to any state previously seen, i.e. being able to choose $\rho_0$ throughout training. This assumption is satisfied for the video games we test as we are able to save and restore simulator states.

**Algorithm 1** Compute Reachability (Exploration Phase of Go-Explore)

```
1: Input: reset priority function $\eta : S \to \mathbb{R}$, total timesteps $T$,
   reset interval $t_{\text{reset}}$, distance threshold $K$
2: Initialize: $S_{\text{disc}} = \emptyset$, set of visited positions
3: for $t = 1, \ldots, T$ do
4:     Observe state $s_t$
5:     if $S_{\text{disc}} = \emptyset$ or $d(S_{\text{disc}}, s_t) > K$ then
6:         $S_{\text{disc}}$.insert($s_t$)
7:     end if
8:     if $t \mod t_{\text{reset}} = 0$ then
9:         Sample state $s_{t+1}$ from $S_{\text{disc}}$ using $\eta$
10:    else
11:        Take random action from $s_t$ to transition to $s_{t+1}$
12:    end if
13: end for
```

## 3. Go-Explore for Simulated 3D Environments

In this section, we describe our adaptation of Go-Explore to 3D video games and the specific reset heuristics we use to identify promising previously discovered locations. Since we do not need the second phase of Go-Explore to imitate an optimal trajectory, the algorithm listed in Algorithm 1 simply builds a set of discovered positions $S_{\text{disc}}$ and thus resembles the first phase of Go-Explore with a tailored reset strategy.

Given a set of 3D positions $X$ and a distance metric $d$, we define the distance from the set $X$ to a point $y$, $d(X,y) := \min\{d(x,y)|x \in X\}$. This can be efficiently computed by data structures such as R*-Trees [23] which support nearest neighbor queries in amortized log time. We define two styles of reset heuristic: one based on state visitation counts, and the other guided by distance to the closest undiscovered point on the nav-mesh.

### 3.1. Visitation-Based.

We discretize the $s = (x,y,z)$ positions in the game to a fixed granularity $K$ and then maintain visitation counts of each state. This allows us to reset to a state in $S_{\text{disc}}$ with probability proportional to $\frac{1}{n(s)}$ where $n(s)$ is the visitation count. By default, we measure distance in meters and use a granularity of $K = 1$, i.e. a new position is deemed to be reached if it is more than one meter away from any previous position. This is in contrast to the cell-based archive chosen by Ecoffet et al. [11] in order to avoid the same physical location being saved multiple times due to viewpoint changes.

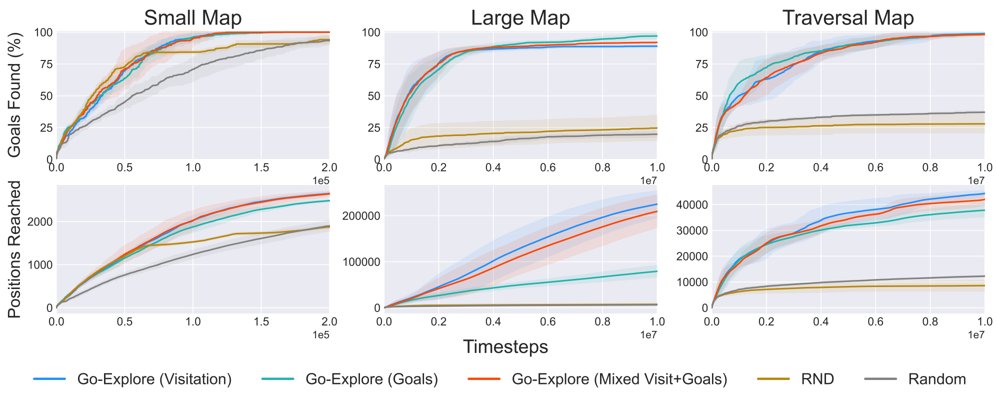

**Figure 2:** Comparative evaluation of Go-Explore against the baselines showing the percentage of nav-mesh goals (illustrated in Figure 1) reached on the top and unique positions discovered (up to 1m discretization) on the bottom. We note that Go-Explore vastly outperforms RND and pure random exploration especially on the larger maps and is reasonably stable to reset heuristic. Results show mean and standard deviation computed over 4 seeds.

### 3.2. Nav-Mesh Goal Guided.

We use the game nav-mesh to produce a set of points $S_{\text{nav-mesh}}$ or "goals" that we expect to be reachable but have not reached yet as a guide to exploration. This set is constructed by sampling points at a pre-specified threshold at initialization. Throughout training, we update $S_{\text{nav-mesh}}$ by draining any goals that are within one meter of a discovered position. We may then reset to discovered states in $S_{\text{disc}}$ with probability proportional to $\frac{1}{d(S_{\text{nav-mesh}},\cdot)}$.

We may also consider a weighted combination of the two reset heuristics as in Ecoffet et al. [11] to define a custom reset priority function $\eta : S \to \mathbb{R}$. This allows agents to balance between the two reset styles and avoid the failure mode of persistently attempting to reset close to an unreachable nav-mesh goal. Once we reset to a promising state, we follow Ecoffet et al. [11] and take random actions to explore. The notion of a goal generated from the nav-mesh could also be extended to goals on any interesting part of the map. For the games we consider, we only rely on the ability to reset to a specified position, the agent's pose and orientation are randomized.

## 4. Experimental Evaluation

We evaluate our approach on proprietary test game levels designed in Unreal Engine 5. We consider three maps of increasing difficulty:

- **Small Map:** A small test bed with obstacles and two intentionally placed bugs (approx. 100m x 100m)
- **Large Map:** A vast cityscape to test the scalability of each algorithm (approx. 1.5km x 1.5km). This map is among the largest considered by modern automated approaches.
- **Traversal Map:** A large multi-level map with challenging geometry as a hard exploration challenge.

The game features 3D voxel observations and optional raycast, position and orientation features. Agents move using a multi-discrete action space matching that of a standard game controller. The maps are illustrated in Figure 1.

For Go-Explore, we evaluate all reset heuristics previously described in the previous section—visitation-based, goal-based and mixed. We use a default distance threshold of $K = 1$ for adding a new point and use the same threshold for evaluating how many unique positions an agent visits over the course of training. The agents are reset every $t_{\text{reset}} = 128$ steps. We first evaluate the performance of Go-Explore against the baselines, RND (using default hyperparameters with a 2-layer voxel CNN, optimized using PPO [24]) and random exploration, across the three test maps. Next, we ablate various components showing that Go-Explore is stable to hyperparameter choice, and that random exploration is critical for strong performance on large maps.

For each experiment, we accelerate training by running 16 independent agents within each map that cannot interact with or see each other. Each agent is synchronized with a central cache of discovered positions. We train for 200K timesteps on the small map and 10M timesteps for both large maps, and repeat each experiment with 4 random seeds. Due to our use of parallelization, wall-clock time for the small map is around 30 minutes, whereas the large maps only take 10 hours to cover on a single F8 Azure Virtual Machine. We use a GeForce RTX 2080 GPU only for the RND baseline.

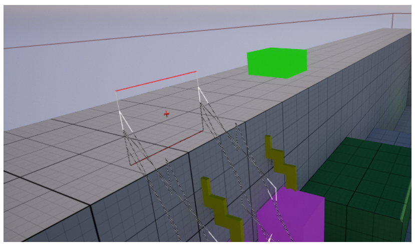
**Figure 3:** Illustration of a bug on the small map surfaced by Go-Explore. The **goal** on the wall means the agent is unexpectedly able to reach the top of the wall and then fall off the map.

## 4.1. Main Evaluation

We first show that our algorithm can discover the intentionally placed bugs on the small map and that it reliably outperforms the baselines across a variety of configurations. Next, on the large maps, the difference between Go-Explore and the baselines grows more stark with the baselines making little progress. On these maps, Go-Explore can cover the vast game worlds and discover many unexpected regions off-map.

### 4.1.1. Bugs on the Small Map.

We compare Go-Explore with all three reset heuristics on the left of Figure 2 evaluating coverage of the nav-mesh goals and unique positions found. On this map, all three reset heuristics are comparable and are strong in terms of positions reached and goals found. We manage to consistently find the two bugs that were placed on the small map, one where an incorrectly configured point led to agents being able to reach the top of the bounding wall and fall off the map which we illustrate in Figure 3 and another where an unreachable volume was incorrectly placed so an agent could incorrectly enter into a solid block. The wall bug can also be identified in Figure 1 where the cluster of yellow points on the south wall of the map are unexpected and then can easily be flagged to the level developer.

### 4.1.2. Large and Traversal Maps.

We perform the same comparison as before on the larger maps as shown in the center and right side of Figure 2. On these maps, the difference between Go-Explore and the baselines grows more stark with Go-Explore being the only algorithm that can reliably cover at least the nav-mesh positions. We also notice a difference between different styles of reset heuristic with Go-Explore. On the large map, whilst the goal-based reset heuristic leads to the quickest coverage of the nav-mesh, it discovers fewer unique positions than the heuristics based on visitation count. This advocates for reset heuristics which can jointly encode information about undiscovered nav-mesh goals and visitation.

In Figure 1, we can see discovered regions of both maps which are far away from the nav-mesh which are unexpected. In the case of the traversal map, these correspond to being able to reach the top of large structures and fall off to unexpected regions. This is a common source of reachability bugs and the automated flagging of problematic areas promises to save level designers and QA significant amounts of time. We additionally note that Go-Explore scales well with map size—when comparing the small and large map, Go-Explore only takes ∼50x more timesteps to cover ∼200x more surface area, when comparing both nav-meshes. Therefore, our algorithm may readily be integrated into nightly continuous testing suites as it *only takes 10 hours on a single machine to fully cover a vast 1.5km x 1.5km game world.* This could be further accelerated with parallel game instances. Go-Explore is also efficient in terms of memory and time as we use a R*-Tree [23] to cache positions. At the end of exploration on the large map, the ∼200K discovered points take around 1MB to store. In a tree this size, checking and inserting a newly discovered point also takes less than a millisecond on average which is far less than the rate of simulation for the game.

**Table 1**
Ablations on the Small Map showing Go-Explore is robust to hyperparameter changes with all setups achieving 100% of the goals before 200K timesteps. The default setting uses a threshold $K = 1\text{m}$ and a reset weight power $p = 1$. Mean and standard deviation shown over 4 seeds.

| Reset Type | Hyper-parameters | Timesteps to All Goals | Positions Found |
|---|---|---|---|
| Goals | Default | 128K | 2485.2 ± 21.4 |
| | $K = 5$ | 110K | 2564.0 ± 7.9 |
| | $p = 1/2$ | 167K | 2430.8 ± 37.6 |
| | $p = 2$ | 106K | 2422.5 ± 14.8 |
| Position | Default | 119K | 2649.5 ± 30.6 |
| | $K = 5$ | **90K** | **2736.0 ± 14.8** |
| | $p = 1/2$ | 156K | 2541.2 ± 21.3 |
| | $p = 2$ | 111K | 2705.8 ± 36.8 |

## 4.2. Ablation Studies

In this subsection, we present ablation studies showing that Go-Explore is stable to hyperparameter choice; and showing the benefit of random exploration over RND even when both have smart resetting. We further show ablations on Go-Explore using uniform sampling to show that the design of reset heuristic is important.

### 4.2.1. Hyperparameters.

We present ablations on the distance threshold $K$ required to add a new point to our set of visited positions $S_{\text{disc}}$ on the small map in Table 1. We also include ablations on the power of the reset heuristic $p$, i.e. resetting using the heuristic $\eta^p$ instead of $\eta$. The original Go-Explore paper used a power of $p = \frac{1}{2}$, i.e. taking the square root of all scores, however we find that $p = 1$ works better for our use case which matches Kolter and Ng [25]. We see a marginal benefit from using $K = 5$ for position based resetting, however we believe $K = 1$ could lead to better checkpointing on hard exploration environments.

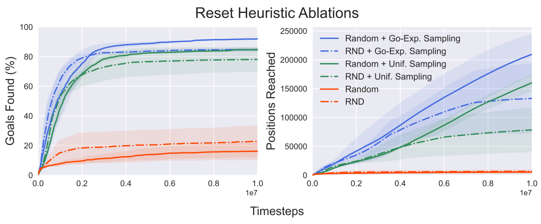

**Figure 4:** Ablations for both random and RND-based exploration under visitation-based sampling (in **blue**) and uniform sampling (in **green**) on the large map. This provides further support for purely random exploration, and shows that well-designed reset heuristics beat naïve uniform sampling. Results show mean and standard deviation computed over 4 seeds.

### 4.2.2. RND vs. Random Exploration.

We show that even when allowing the RND agent to reset to promising positions in the same way as Go-Explore on the large map, labeled "RND + Go-Exp. Sampling" in Figure 4, the agent still under-performs Go-Explore (Random + Go-Exp. Sampling) in terms of final performance. However, we do note that RND-based exploration (in dashed lines) does tend to be initially better but then trails off. This could imply that the learned exploratory behavior from RND does not transfer to later regions of the map, perhaps preferring areas it has obtained reward from in the past. This lends further support to our use of Go-Explore style *random exploration* which requires no neural network overhead and is *simpler to implement.*

### 4.2.3. Go-Explore Heuristics vs. Uniform Sampling.

We further analyze the design choices involved in selecting sampling heuristics and ask how much better they are than just sampling visited points uniformly at random. This corresponds to the comparison between Go-Exp. Sampling (in green) vs. Unif. Sampling in Figure 4 (in blue). Although uniform sampling hugely boosts performance over simply resetting from the start position (in red), it does not reach the same levels of performance as with the carefully chosen heuristics.

## 5. Related Work

Go-Explore has seen extensive use in discovering optimal policies for 2D games like Atari [11]; but to our knowledge, it has not seen use in reachability testing for large-scale modern 3D game environments.

### 5.1. Learning to Explore.

Curiosity and imitation-based approaches where agents learn to explore have been successfully used to automate game testing in past work. Gordillo et al. [8] showed that a count-based exploration bonus with additional bonuses for reaching corners of the map approach with visitation-based resets was successful in covering a 500m x 500m map. In contrast, we don't need a reward signal and our resets happen at a much higher frequency. CCPT [9] used a hybrid approach combining RND and imitating human trajectories. However, imitation-based approaches are liable to break with large map changes during development. Inspector [10] combines RND with a separate module that attempts to identify and interact with key objects in a scene. The interaction module could be a promising orthogonal extension to Go-Explore. Bergdahl et al. [26] consider an approach where an agent is reward on reaching pre-specified goals which inherently biases against exploring other parts of the game.

### 5.2. Graph-based Search.

Traditional search-based methods have shown promise in small games but are harder to scale. Rapidly-Exploring Random Trees (RRT, Zhan et al. [27]) grows a state tree and labels edges with actions; but scales with the game action space. CA-RRT [28] improves on RRT by augmenting search with human demonstrations but is still only evaluated on relatively small maps. SPTM [29] navigates by building a graph of the environment and learning to retrieve nodes for planning.

## 6. Conclusion

In this paper, we show that Go-Explore allows for simple, scalable and efficient reachability testing of the vast 3D game maps typically found in modern AAA video games without the need for any human demonstration or knowledge of the game dynamics. In contrast to prior curiosity-based RL approaches, Go-Explore avoids the issue of catastrophic forgetting by maintaining a cache of discovered positions. The random exploration advocated by Ecoffet et al. [11] is significantly simpler to implement and avoids the use of neural networks. By flagging discovered points that are far away from the nav-mesh, we are able to identify reachability issues and highlight problematic points on the map. Due to the fast run-time and ease of implementation of the algorithm, we expect Go-Explore may be readily integrated into a continuous testing framework which would allow bugs to be identified as soon as each level is updated.

## Acknowledgments

We would like to thank Sam Devlin, Anssi Kanervisto, Tabish Rashid, Tim Pearce, Mingfei Sun, and John Langford for reviewing earlier drafts of this work and many insightful discussions.

## References

[1] A. Nantes, R. Brown, F. Maire, A framework for the semi-automatic testing of video games, in: Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, volume 4, 2008.

[2] E. Bulut, Playboring in the tester pit: The convergence of precarity and the degradation of fun in video game testing, Television & New Media 16 (2015). doi:10.1177/1527476414525241.

[3] C. Lewis, J. Whitehead, N. Wardrip-Fruin, What went wrong: a taxonomy of video game bugs, in: Proceedings of the fifth international conference on the foundations of digital games, 2010.

[4] A. Truelove, E. Santana de Almeida, I. Ahmed, We'll fix it in post: What do bug fixes in video game update notes tell us?, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021. doi:10.1109/ICSE43902.2021.00073.

[5] A. M. Albaghajati, M. A. K. Ahmed, Video game automated testing approaches: An assessment framework, IEEE Transactions on Games (2020).

[6] S. Ariyurek, A. Betin-Can, E. Surer, Automated video game testing using synthetic and humanlike agents, IEEE Transactions on Games (2021). doi:10.1109/TG.2019.2947597.

[7] C. Politowski, F. Petrillo, Y.-G. Guéhéneuc, A survey of video game testing, in: 2021 IEEE/ACM International Conference on Automation of Software Test (AST), 2021. doi:10.1109/AST52587.2021.00018.

[8] C. Gordillo, J. Bergdahl, K. Tollmar, L. Gisslén, Improving playtesting coverage via curiosity driven reinforcement learning agents, 2021. doi:10.48550/ARXIV.2103.13798.

[9] A. Sestini, L. Gisslén, J. Bergdahl, K. Tollmar, A. D. Bagdanov, Ccpt: Automatic gameplay testing and validation with curiosity-conditioned proximal trajectories, 2022. doi:10.48550/ARXIV.2202.10057.

[10] G. Liu, M. Cai, L. Zhao, T. Qin, A. Brown, J. Bischoff, T.-Y. Liu, Inspector: Pixel-based automated game testing via exploration, detection, and investigation, ArXiv abs/2207.08379 (2022).

[11] A. Ecoffet, J. Huizinga, J. Lehman, K. Stanley, J. Clune, First return, then explore, Nature 590 (2021) 580–586. doi:10.1038/s41586-020-03157-9.

[12] Atari vcs/2600 easter egg list, 2018., ???? URL: http://www.ataricompendium.com/game_library/easter_eggs/vcs/easter_eggs.html.

[13] Y. Burda, H. Edwards, A. Storkey, O. Klimov, Exploration by random network distillation, in: International Conference on Learning Representations, 2019.

[14] G. Snook, Simplified 3d movement and pathfinding using navigation meshes, in: Game Programming Gems, Charles River Media, 2000.

[15] P. Tozour, I. Austin, Building a near-optimal navigation mesh, AI game programming wisdom (2002).

[16] F. Dunn, I. Parberry, 3D Math Primer for Graphics and Game Development, second ed., A K Peters/CRC Press, Boca Raton, FL, 2011.

[17] P. E. Hart, N. J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Transactions on Systems Science and Cybernetics 4 (1968). doi:10.1109/TSSC.1968.300136.

[18] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. A. Riedmiller, Playing atari with deep reinforcement learning, CoRR abs/1312.5602 (2013). arXiv:1312.5602.

[19] R. S. Sutton, A. G. Barto, Reinforcement Learning: An Introduction, second ed., The MIT Press, 2018.

[20] G. Ostrovski, M. G. Bellemare, A. van den Oord, R. Munos, Count-based exploration with neural density models, in: Proceedings of the 34th International Conference on Machine Learning, PMLR, 2017.

[21] M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, R. Munos, Unifying count-based exploration and intrinsic motivation, 2016. doi:10.48550/ARXIV.1606.01868.

[22] A. Hussein, M. M. Gaber, E. Elyan, C. Jayne, Imitation learning: A survey of learning methods, ACM Computing Surveys (CSUR) 50 (2017).

[23] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The r*-tree: An efficient and robust access method for points and rectangles, in: Proceedings of the 1990 ACM SIGMOD International Conference on

Management of Data, 1990.

[24] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, Proximal policy optimization algorithms, 2017. doi:10.48550/ARXIV.1707.06347.

[25] J. Z. Kolter, A. Y. Ng, Near-bayesian exploration in polynomial time, in: Proceedings of the 26th Annual International Conference on Machine Learning, 2009.

[26] J. Bergdahl, C. Gordillo, K. Tollmar, L. Gisslén, Augmenting automated game testing with deep reinforcement learning, in: 2020 IEEE Conference on Games (CoG), 2020. doi:10.1109/CoG47356.2020.9231552.

[27] Z. Zhan, B. Aytemiz, A. M. Smith, Taking the scenic route: Automatic exploration for videogames, 2018. doi:10.48550/ARXIV.1812.03125.

[28] M. Zuo, L. Schick, M. Gombolay, N. Gopalan, Efficient exploration via first-person behavior cloning assisted rapidly-exploring random trees, 2022. doi:10.48550/ARXIV.2203.12774.

[29] N. Savinov, A. Dosovitskiy, V. Koltun, Semi-parametric topological memory for navigation, in: International Conference on Learning Representations, 2018.