Graph augmented Deep Reinforcement Learning in the GameRLand3D environment

Anonymous Authors

Afiliation

Abstract

We introduce a set of hybrid graph and Deep Reinforcement Learning approaches for planning and navigation in 3D video games featuring complex environments with disconnected regions reachable by agents using special actions. We also present "GameRLand3D", a new benchmark and soon to be released environment built with the Unity engine able to generate complex procedural 3D maps for navigation tasks. We quantify the limitations of end-to-end Deep RL approaches in vast environments and explore hybrid techniques that combine a low level policy and a graph based high level classical planner. In addition to providing human-interpretable paths, the approach improves the generalization performance of an end-to-end approach in unseen maps and achieves a 20% absolute increase in success rate over a recurrent end-to-end agent on a point to point navigation task in maps of size 1km$\times$1km.

Introduction

Long term planning and navigation is an important component when looking to realistically control a non-player character (NPC) in a video game. Whether the NPC is player facing, in order to enhance a human players experience, or used for automated testing to find bugs and exploits, the ability to navigate in a complex environment is paramount and is the foundation of every NPC. In the field of video games the Navigation Mesh (NavMesh) (Snook 2000) has become the ubiquitous approach for planning and navigation. In recent years, the sheer scale of video game environments have highlighted the limitations of the NavMesh, particularly in games with complex navigation components such as double jumping, teleportation, jump pads, grappling hooks, and wall runs. Given the successes of Deep Reinforcement Learning (RL) in domains of robotic control and Atari video games, recent works have looked to exploit these approaches for navigation in large 3D video game environments. These works have shown promise in smaller game worlds, but have highlighted that as environments scale to hundreds of thousands of square metres, the limitations of end-to-end learning-based methods are encountered.

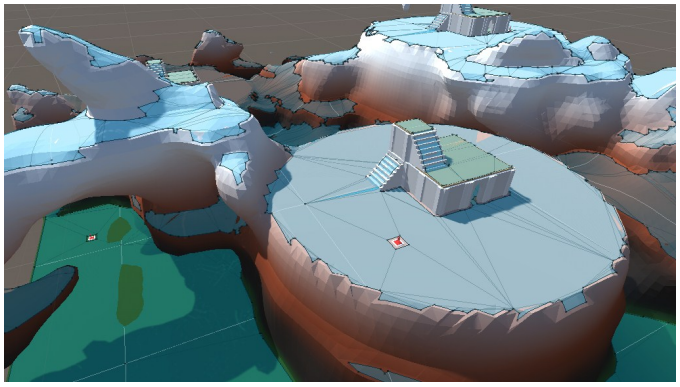

Figure 1: The proposed hybrid planner is capable of performing real-time navigation in complex 3D gaming environments featuring geometrically disconnected regions reachable by agents through special actions. These environments are difficult for approaches based on the NavMesh.

It is for these reasons that we have developed the GameRLand3D environment, to quantify the performance of end-to-end approaches and to explore potential solutions. In addition to the release of the open source GameRLand3D environment, we introduce a new hybrid navigation method, which combines a recurrent baseline agent with a graph-based classical planner. The resulting hierarchical planner combines low level waypoint-based control and navigation with interpretable high level plans.

Currently, with few exceptions, navigation in video games relies on the NavMesh, which is created and used as follows.

1. A graph representation of the world is pre-generated from the game geometry.
2. At runtime a pathfinding algorithm such as A* (Hart, Nilsson, and Raphael 1968b) is run on this graph to find the shortest path between two locations in the game.
3. A player controller, attached to the character is used for way-point based navigation to follow the sequence of way-points returned by the planner.

While the navigation mesh provides a compact representation of the world, it is somewhat limited and results in disconnected regions in complex environments, demonstrated

in Figure 1. The disconnected regions are due to some assumptions of the underlying NavMesh generation algorithm, where the user provides the maximum inclination and height the agent can walk and jump, but other actions such as double jumping or using a jump pad are not taken into account. Connections due to special character abilities such as jump pads, grappling hooks and teleporters often involve time consuming manual editing of the graph.

Recent works have applied an end-to-end Deep Reinforcement Learning (RL) approach to address this problem (Alonso et al. 2021) and have shown that when trained and evaluated on small (50m×50m) to medium maps (100m×100m) maps, end-to-end approaches achieve good performance. However, when evaluated in large (300m×300m) environments, policies learned using RL perform well on near and medium distance goals but poorly on farther challenging goals. Euclidean distance is not the best measure of complexity in this case, but as the odds of encountering a hard obstacle increase with the size of the map, it is a decent proxy for the complexity of navigation.

In this paper we explore a hybrid approach that aims to combine the strengths of the classical graph-based NavMesh approach with the capacity of end-to-end RL to incorporate complex actions and abilities to improve navigation performance. Our Graph augmented RL agent outperforms the generalization performance of a learning-based agent in maps of 250m×250m and greatly improves the performance of a map specific agent that is trained and evaluated on a game world of 1km×1km, 10 times larger than those typically used in the research community (Alonso et al. 2021). Our contributions are as follows:

- A novel hybrid graph and RL approach that decouples long distances planning and low level navigation and outperforms end-to-end Deep RL methods in both a generalization and map specific setup. Notably, this approach provides interpretable plans which can be adjusted by a game designer without the need to retrain the RL agent.
- The release of an open-source 3D procedural environment in Unity which can generate complex 3D worlds of more than one million square meters.
- We quantify the limitations of the performance of end-to-end Deep RL approaches in large environments.
- A detailed empirical evaluation and ablation of 7 hybrid RL and graph based approaches, with a special focus on their run time performance, a key requirement of video games operating in real time at 60 frames per second.

Related work

Deep RL environments

There has been a proliferation of new Deep RL environments in the last five years, the most well known being the gym framework (Brockman et al. 2016) which provides an interface to fully observable games such as the Atari-57 benchmark and continuous control tasks. There are a variety of partially observable 3D simulators available, starting with more game-like environments such as the ViZDoom simulator (Wydmuch, Kempka, and Jaśkowski 2018), DeepMind-Lab (Beattie et al. 2016) and Malmo (Johnson et al. 2016). A number of photo-realistic simulators have also been released such as Gibson (Xia et al. 2018) and AI2Thor (Kolve et al. 2017), the datasets that these simulators use are also available in the highly efficient Habitat-Lab simulator (Manolis Savva et al. 2019), in addition to a number of benchmark 3D navigation tasks. The aforementioned environments were designed to train and evaluate RL agents in relatively small environments of the order of hundreds of square metres, with relatively small action spaces. Moreover, they do not consider the complex actions that are in modern video games, such as double jumping, teleportation, jump pads, grappling hooks, and wall runs. This is the motivation behind the GameRLand3D environment.

Classical planning and navigation in video games

Existing methods for path planning in the video game industry mostly rely on building a Navigation Mesh (NavMesh)(Snook 2000). A NavMesh is a 2D/3D graph whose nodes represent convex polygonal regions of the walkable space, typically triangles, and whose edges indicate possibilities of navigation between regions. To find a path between two locations in the world, one can then apply path finding algorithms like A* (Hart, Nilsson, and Raphael 1968a) or one of its variants on this graph. This classical approach is robust for many applications but presents some limitations. In particular, when it is possible for the agent to use mobility actions (like dashing, jumping, double jumping, using a jetpack), the high number of places it can reach from each location dramatically increases the connectivity of the NavMesh (Alonso et al. 2021; Gordillo et al. 2021). Therefore, it increases its memory cost as well as the runtime cost of the pathfinding algorithms, to the point where using a NavMesh ends up being too costly, constraining game design. Another important aspect that a NavMesh does not account for is the kind of constraint that a character can have, such as the turn radius of a car or a plane in 3D. This kind of constraint can be impossible to encode in a NavMesh in an efficient manner. NavMesh based approaches have also been extended to dynamic environments (van Toll, Cook IV, and Geraerts 2012).

Neural approaches to navigation

The field of Deep Reinforcement Learning (RL) has gained attention with successes on board games (Silver et al. 2016) and Atari games (Mnih et al. 2015). Recent works have applied Deep RL for the control of an agent in 3D environments (Mirowski et al. 2016; Jaderberg et al. 2016), exploring the use of auxiliary tasks such as depth prediction, loop detection and reward prediction to accelerate learning. Other recent work uses street-view scenes to train an agent to navigate in city environments (Mirowski et al. 2018).

To infer long term dependencies and store pertinent information about the partially observable environment, network architectures typically incorporate recurrent memory such as Gated Recurrent Units (Chung et al. 2015) or Long Short-Term Memory (Hochreiter and Schmidhuber 1997). Extensions to memory based neural approaches began with Neural Turing Machines (Graves, Wayne, and Danihelka 2014) and

Differentiable Neural Computers (Graves et al. 2016) and have been adapted to Deep RL agents (Wayne et al. 2018).

Spatially structured memory architectures have been shown to augment an agent's performance in 3D environments and are broadly split into two categories: metric maps which discretize the environment into a grid based structure and topological maps which produce node embeddings at key points in the environment. Research in learning to use a metric map is extensive and includes spatially structured memory (Parisotto and Salakhutdinov 2017), Neural SLAM based approaches (Zhang et al. 2017) and approaches incorporating projective geometry and neural memory (Gupta et al. 2017; Bhatti et al. 2016). These techniques are combined, extended and evaluated in (Beeching et al. 2020b; Wani et al. 2020). Research combining learning, navigation in 3D environments and topological representations has been limited in recent years with notable works being (Savinov, Dosovitskiy, and Koltun 2018) who create graph through random exploration in ViZDoom RL environment (Wydmuch, Kempka, and Jaśkowski 2018). (Eysenbach, Salakhutdinov, and Levine 2019) also performs planning in 3D environments on a graph-based structure created from randomly sampled observations, with node distances estimated with value estimates. (Beeching et al. 2020a) perform topological planning with a neural approximation of a classical planning algorithm that can be applied in uncertain environments. Other recent approaches such as (Chaplot et al. 2020) build a graph structure, but rely on 360 degree camera measurements. Practically all of the aforementioned approaches are applied in small planar environments of hundreds of square metres, and scaling these methods to the environment where we apply our hybrid approach poses several challenges. Availability of ground truth: most graph based methods require ground truth actions and graph connectivity, as the calculation of optimal actions is often intractable in 3D environments of hundreds of thousands of square metres this precludes this possibility. We are also constrained due to inference time: many of these methods do not operate at real time speed, particularly when controlling hundreds of agents in parallel.

Environment and task definition

In order to address the challenging problem of navigation in vast complex 3D environments we have developed the GameRLand3D environment. GameRLand3D allows the creation of enormous worlds and enables complex interactions such as double jumping and jump-pads, includes buildings, water and lava hazards. The environment is built using the Unity-ML agents (Juliani et al. 2018) framework, which provides a versatile tool to build RL environments. We interact with the environment by controlling an agent, represented as a 1.8 meter humanoid character, shown in Figure .

Our environment implementation uses 3D Perlin noise (Perlin 1985) for the procedural generation of the game world, followed by random placements of buildings, plateaus and jump pads to create a challenging state space for the agent to navigate. Readers can refer to the appendix for further details.

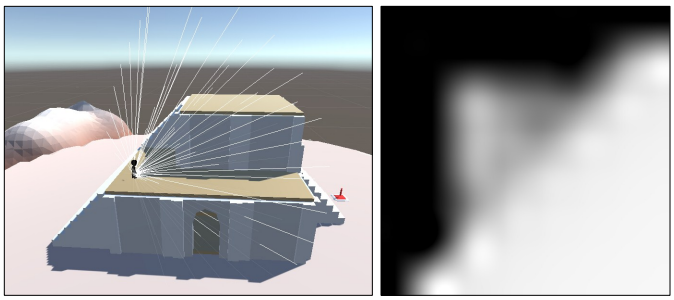

Figure 2: Left: An agent's depth observation in the GameRLand3D environment. Right: the resulting 2D observation in tensor form, that will be processed by the agent's CNN.

Task The task we consider is point to point navigation in large, procedurally generated environments, where the agent is spawned at a location p_{spawn} in the environment and must navigate to a goal location p_{goal}. Start and goal locations are generated using a NavMesh for simplicity, through a function $f_{navmesh}$ from which we can sample valid navigation locations. The NavMesh contains many disconnected regions and cannot be directly used for long distances navigation. To reduce overall inference time and to allow for planning over longer time horizons, we request a decision from the agent's policy network every 10 time-steps, repeating the actions for the intermediary 10 steps.

Constraints Video game environments provide certain advantages for training RL agents when compared to learning a model that may be transferred to the real world. The position and orientation of the agent are known and we are not expected to rely on noisy estimates and fulfill safety guarantees which add an additional challenge to robotic navigation. Video games do have other requirements and constraints, model implementations must be highly optimized, parallelized and run at real time speeds, making decisions at a minimum of 60 times per second. If an agent is player facing, its behavior must be credibly "human-like", a subjective metric that is discussed in (Devlin et al. 2021). Finally, methods should provide interpretable information, so that game designers can be confident about adopting it in production.

Observations Rendering agent observations from the environment with a virtual monocular camera is prohibitively expensive for NPC control in video games, so the agent's observations are a depth measurement of a 8×8 cone of raycasts in front of the agent (see figure). We also provide useful information such as the location of the goal in the agent's frame of reference, the agent's velocity and acceleration.

Actions The environment actions are continuous and correspond to actions that are typically available to a human player: forward, backward, strafe, turn, jump and double-jump (where a player can execute a second jump while in mid-air). The jump action is treated like a continuous action for the RL policy and is discretized in the environment.

Rewards Similar to (Alonso et al. 2021), We combine three rewards: A **sparse reward** of +1 when the agent suc-

ceeds in the task and -1 when the agent fails. A **dense reward** every step in which the agent reduces its best euclidean distance to the goal location, divided by a normalization factor λ. A **time-step penalty** of -0.0005 per step, to encourage the agent to perform the task quickly. An episode is terminated when the agent is within 1m of its goal, the agent falls off the map or the agent has not decreased its best euclidean distance to the goal for 300 time-steps.

Baseline models

The baseline policy was trained using a recurrent version of a sample efficient off-policy actor critic RL algorithm, Soft Actor Critic (SAC) (Haarnoja et al. 2018a), with a learned entropy coefficient and no state value network (Haarnoja et al. 2018b). The SAC algorithm aims to finds a policy that maximizes cumulative reward and the entropy of each visited state.

$$\pi^* = \arg\max_{\pi} \sum_{t} E_{(\mathbf{s}_t, \mathbf{a}_t) \sim \rho_\pi} \left[r\left(\mathbf{s}_t, \mathbf{a}_t\right) + \alpha \mathcal{H}\left(\pi\left(\cdot \mid \mathbf{s}_t\right)\right) \right] \tag{1}$$

We include a learnable entropy temperature parameter τ, to enable automatic optimization of the entropy of the objective function. We found that having independent policy and Q-network embeddings improves the reliability of the Q-value estimates and led to more robust, stable training. We trained several versions of the baseline agent architecture. A **reactive** agent that receives the 8x8 raycast observation and a **memory-based** recurrent agent that extends the reactive agent with an LSTM in order to be able to learn longer term dependencies. We use a burn-in to initialize the hidden state as recommended in (Kapturowski et al. 2018; Paine et al. 2019). The reader can refer to the appendix for details of the architecture of the baseline model.

In order to estimate the loss in performance when a policy is learned on one specific map or when a policy is learned on a set of maps and transferred to another, we trained two versions of the memory based agent: a general agent and a map specific agent. The general agent is trained on a large dataset of 128 procedurally generated maps. During training we evaluate the performance of the policy on a set of 4 validation maps and keep the model with the highest success rate. In addition, independently we train 4 map specific agents, one per test map, the objective being to quantify the decrease in performance, or *generalization gap*, between the map specific and general agent, see Figure 3. The graph augmented methods that we discuss in the following section use the general baseline for point to point navigation and perform similarly to the map specific agent. We note that a general agent is a crucial requirement in the video game production setting, as it removes the need of retraining on every new map instance.

Graph augmented RL agents

In order to improve the performance of the map general baseline agent, we explored several methods that augment the baseline policy with a directed graph (digraph) for planning and way-point to way-point navigation. This approach

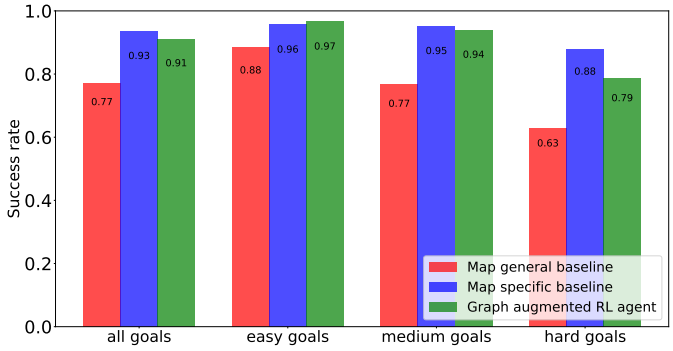

Figure 3: The generalization gap in performance between a map specific agent trained on a single map and a general agent trained on 128 maps and then evaluated on the same unseen map. Compared against one of the proposed Graph augmented RL agents, that are introduced in this paper.

decomposes the problem of navigation over long distances into sequences of smaller sub-problems. The motivation is that the agent performs well on short goals. By breaking down a long distance goal into a sequence of easy, short term goals, we can improve the success rate of the agent, provide interpretable paths and allow for identification of disconnected regions in the environment topology.

For all methods, we build a graph G comprised of a set of vertices V and edges E, where edge $e(i, j)$ connects vertex v_i to vertex v_j. Each edge has a cost $c(i, j)$. Assigned to each vertex is a 3D position in the game. Planning from a position p_{start} to a goal position p_{goal} is performed by identifying the nearest vertex in the graph to the starting position v_{start} and the nearest vertex to the goal position v_{goal}. The optimal sequence of vertex locations to traverse in order to reach v_{goal} is found with the classical path planning algorithm Dijkstra (Dijkstra 1959).

The process of graph creation follows three key steps: identify vertex locations, testing edge connectivity and graph building. The following sections describe the different methods we explored for identifying the vertex locations, edges, edge costs and building the graph. Each method provides a compromise between run-time performance, graph construction time and overall success rate. As user requirements may differ from one game to the next, we provide analysis of each approach in the experiments section .

Constrained and unconstrained vertex sampling

Unconstrained random vertex placement In the unconstrained random approach we sample n vertices from the function $f_{navmesh}$. This provides vertices that are located on navigable terrain, but the edge connectivity is unknown. Edge connectivity is computed by first identifying the k nearest neighbors of each vertex and then evaluating a short point goal episode between each vertex and its k-nearest neighbors. If the episode is successful then the directed edge is considered to be connected with its cost equal to the number of steps in the episode. Using the agent steps as a cost over the euclidean distance leads to graph planning that

takes into account local obstacles in the planning process.

Distance constrained random vertex placement We augment the unconstrained random approach with coverage criteria, in order to spread nodes evenly across the environment and ensure that we can estimate connectivity all over the map. We aim to identify the set of vertex positions V that satisfy the criteria $d_V := \max_{x,y \in V, x \neq y} d(x, y)$, where $d(x, y)$ is the euclidean distance between two vertex positions. We find the set of positions V using a greedy, non-optimal, strategy by first calculating the optimal average distance between vertices, assuming the environment was a planar surface and then sampling vertex locations from $f_{navmesh}$ until no nodes are present in the proximity of the sampled vertex location. In order to avoid infinite loops this test is repeated 100 times and after each test the distance criterion is decayed by a factor of 0.98.

Flooding approaches

This approach aims to produce a dense graph that evenly covers the topology of the environment. The dense coverage of vertices should provide a robust estimation of navigability with the downside of increased run-time for graph building and search. We were inspired by NavMesh construction algorithms in video games (Snook 2000), where a surface of connected locations is iteratively calculated from a starting point. We initialize with a parent seed location p_{seed}, which spawns $k \times h$ child vertices evenly distributed on a sphere of radius d_{flood}. To ensure new locations follow the topology of the environment, we identify the nearest navigable surface in the environment with two raycasts in the vertical axis. Directed edge candidates are then evaluated by rolling out the baseline policy, successful edges become seeds for the next round of flooding. Seeds which create duplicate edges which share the same start and end location are rejected. When the flooding processes completes, we implemented a check for disconnected regions by sampling from $f_{navmesh}$ and performing a nearest neighbor check. If there are no neighbors within $2 * d_{flood}$, we seed from the sampled location. Figure 4 shows an example of 4 steps in the flooding method.

One weakness of this approach is it can be sensitive to the size of d_{flood}. While smaller distances lead to denser coverage, they can result in disconnected regions as larger gaps that exceed d_{flood} will not be tested for connectivity. In order to identify and connect disconnected regions, we also consider a post-process where for each node we identify its potential grandchildren, vertices within $2 * d_{flood}$. We then perform a graph search and compare the path length to the grandchild vertex to the euclidean distance to the grandchild vertex. If no path is found or the ratio of the distances is greater than 5, we run an edge connectivity test between the vertex and the grandchild. This step increases the success rate of this method with the downside of increased run-time, discussed further in the experiments section.

Trajectory based

Real world multiplayer video games often collect trajectories of player data: for analysis, game balancing and debugging. We use trajectories such as these to build a graph for planning and navigation. We have implemented and evaluated several trajectory based methods in order to place vertices in key locations on the map, such as doors, bridges, jump pads and choke-points. An assumption we make is that trajectories are available that have been sampled from the evaluation environments. These trajectories can be either player data, found in abundance in live multiplayer games, or trajectories sampled from rolling out the baseline agent on random start and end goal locations.

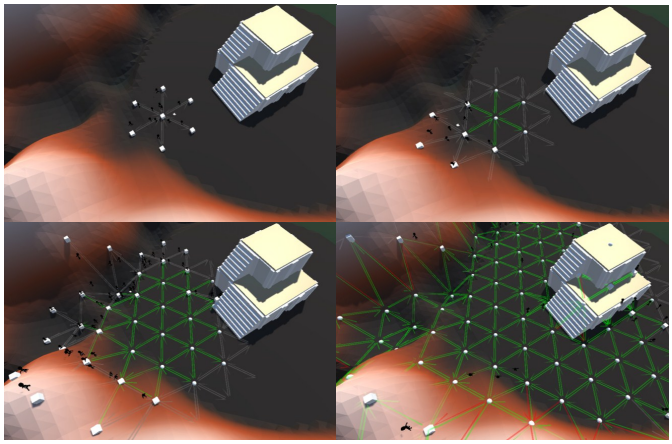

Figure 4: Initial steps of the flood method for building a dense graph which covers the topology of the environment.

Trajectory merging Given a set of trajectories $\{\tau_0, ..., \tau_n\}$, we consider each trajectory as a single directed graph with vertices corresponding to the positions in the trajectory and directed edges between sequential vertices. To induce sparsity we select vertices every d_{node_dist} meters. Key vertices and edges are identified by iteratively merging nearby vertices and edges that are within a distance d_{merge}. In our other approaches, the number of vertices in the graph was directly controlled by a hyper-parameter, however, in this case the number of vertices is controlled indirectly by varying the number of trajectories and the trajectory merging distance. In order to perform nearest neighbor search in $\log(n)$ time, we use a KDTree (Bentley 1975). Once the key vertices have been identified, edge building can be done in a number of ways:

- **Trajectory edges** keeps the directed edges that are sampled when the trajectories were parsed to identify the key vertices. This is quick to run, but may suffer from over-optimistic planning as discussed in section .
- **Nearest neighbor edge tests** performs the edge connectivity tests with the k nearest neighbors, similarly to the random vertex placement methods.

Density based vertex placement In the density estimation approach the objective is, given a set of trajectories, to estimate a probability density function (PDF) $P_{vertex}(x, y, z)$ that identifies key regions where there is high agent footfall, in order to sample vertex locations from the PDF. We estimate the PDF using a count based system,

Method	Success rate			
	All goals	Easy goals	Medium goals	Hard goals
Map specific LSTM baseline (not comparable)	0.899 ± 0.000	0.911 ± 0.000	0.891 ± 0.001	0.892 ± 0.001
General reactive baseline	0.680 ± 0.000	0.821 ± 0.000	0.619 ± 0.000	0.574 ± 0.000
General LSTM baseline	0.696 ± 0.000	0.833 ± 0.000	0.661 ± 0.001	0.556 ± 0.002
Unconstrained random vertex placement	0.824 ± 0.023	0.878 ± 0.013	0.812 ± 0.029	0.767 ± 0.029
Distance constrained random vertex placement	0.859 ± 0.002	0.904 ± 0.002	0.850 ± 0.004	0.809 ± 0.004
Density based vertex placement	0.833 ± 0.004	0.888 ± 0.007	0.820 ± 0.004	0.776 ± 0.006
Trajectory merging	0.812 ± 0.010	0.880 ± 0.010	0.808 ± 0.010	0.724 ± 0.009
Trajectory merging vertices with edge tests	0.862 ± 0.002	0.903 ± 0.004	0.855 ± 0.001	0.816 ± 0.002
Flood	0.800 ± 0.027	0.822 ± 0.040	0.804 ± 0.023	0.765 ± 0.020
Flood with grandchild edges	0.856 ± 0.008	0.898 ± 0.004	0.849 ± 0.012	0.808 ± 0.010

Table 1: Evaluation on a test set of 4 maps of size 250m×250m. In addition to general reach and memory-based policies, we include the performance of 4 map specific policies, learned independently on each test map. We compare against the best performance of the 7 graph-based hybrid methods.

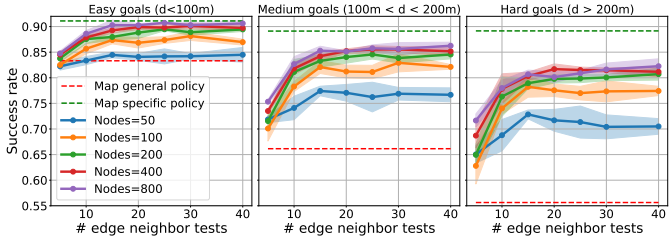

Figure 5: Results for different hyper-parameters for the method *Distance constrained random vertex placement*: graph sizes of 50 to 800 nodes; edge candidates of 5 to 40 neighbors

where we first seed the environment with a dense coverage of potential vertex locations, by sampling from $f_{navmesh}$. For every location in each trajectory we increment a counter on the nearest neighbor vertex. We use a frequentist approach to calculate the final probability distribution and sample a subset of k vertex locations from the dense population as potential vertex locations. A temperature hyper-parameter T controls the uniformity of the PDF.

Optimizations

Robust estimation of edge connectivity One weakness of performing the edge connectivity test once is that there is a small probability that agents can introduce an edge in the graph that often fails, leading to overconfident planning. This problem is compounded when there are several edges in the graph that are difficult to navigate. In order to identify and prune these edges, we evaluate an option to repeat the edge connectivity tests several times and keep the edges where the percentage of the tests exceed a given threshold. This approach leads to robust plans and higher performance, with the downside of a longer graph building step. An analysis of the increase in empirical performance is shown in the supplementary material.

Re-planning Qualitative analysis demonstrated that occasionally a graph augmented agent would fail to reach its next way-point, resulting in an agent that was often stuck as it had fallen off the side of an obstacle. This was resolved with the implementation of a re-planning step, where if the agent does not decrease its distance to the next way-point for 50 time-steps we automatically re-plan from its current location to the goal location. We observed that re-planning increased the success rate of all the examined methods.

Experiments

Setup We generate maps of size 250m×250m×80m, 128 training maps, 4 validation maps and 4 tests maps to test the initial implementation of the baseline and graph augmented methods. We then scale this approach to maps of 1km×1km ×80m to evaluate the performance in vast game worlds. Such a map is notably larger than the largest maps tested in the literature (Alonso et al. 2021).

While other works also include the Success weighted by Path Length (SPL) metric (Anderson et al. 2018), we are unable to compare against this metric as the SPL metric requires ground truth shortest paths which cannot be computed in such large environments with a complex action spaces. We separate the goals into three groups, easy, medium and hard, based on distance between the start and end points to break down the performance of each method. The results on hard, long distance goals, are of particular interest as this is where the baseline policy had the worst performance.

Hyper-parameter optimizations We extensively optimized the hyper-parameters of each approach and report the results of the best performing configuration for each technique, with the exact values and ablations detailed in the supplementary material. Each test was evaluated with three seeds in order to quantify the average and variation of performance that can be achieved with these methods. We evaluate the performance of each method using the success rate metric on 4 test maps, with 2,500 goals per map. As an example, we report in Figure 5 the evaluation of a single method, *Random vertex with coverage constraint*, in numerous hyper-parameter configurations. In this case we have varied the number of nodes and the number of nearest neighbors with which to perform edge connectivity tests. Displays for the other methods with various hyper-parameter configurations are included in the supplementary material.

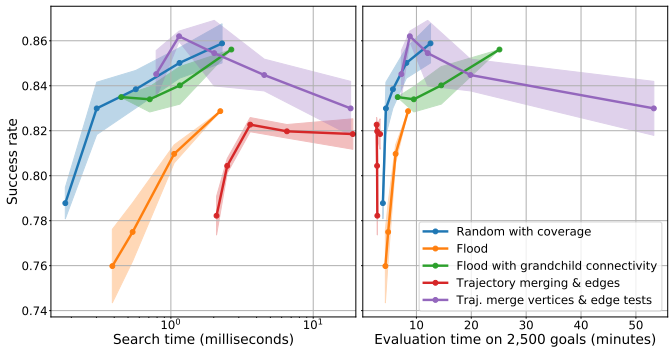

Figure 6: Run-time trade-offs of the graph augmented approaches. Left: Average graph search time in ms for each method in several configurations. Right: Average graph construction and evaluation time in minutes on 2,500 goals.

Comparisons We evaluate the performances of the proposed graph augmented methods, in the different variants according to the chosen graph creation approaches, and compare them to the performance of the recurrent baselines, shown in table 1. The method that achieves the highest success rate is that of the trajectory-based approach with edge connection tests, with a success rate of 86.2%, an improvement of 16.6% over the end-to-end baseline. This method identifies key vertex locations and has the advantage of performing repeated edge connectivity tests for more robust connectivity estimation and planning. In the case of a new map, where player or agent trajectories are not available, the *Distance Constrained Random Vertex Placement* and *Flood with Grandchild Edges* approaches achieve comparable success rates of 85.9% and 85.6%, respectively.

Run-time analysis Run-time performance is critical in real time video games, where the game logic and rendering must be executed 60 times per second. We perform an analysis of the run-time performance of the various approaches. In particular, we are interested in the time it takes to build the graph, to evaluate 2,500 goals on a test map and average duration of a graph search when exploiting the Hybrid approach. Figure 6 summarizes the results for key sets of hyper-parameters from each technique. We observe that there is a trade-off, with graph trajectory based methods being faster to build, but achieving poorer performance. The optimal choice therefore depends on the exact use case — for an offline exhaustive test of a map, the longer running random or flood methods would be more suitable, whereas if the use-case requires a graph to be built in near real time, the trajectory based methods may be more suitable.

Scaling to production size maps Previous experiments have focused on generalization of a learned policy to unseen maps. We have extended our method to a vast 1km×1km×200m map. We find that even a map specific baseline RL agent, whose policy is learned directly on the map it is evaluated on (environment overfit), performs poorly compared to our hybrid agent, with an overall success rate of 72%. We augment the map specific agent with the

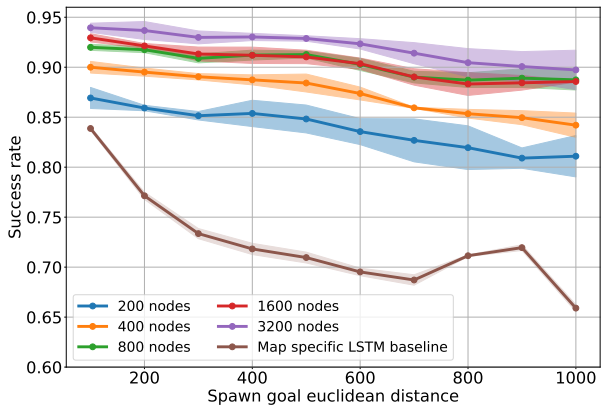

Figure 7: Evaluation of the Random Graph with coverage approach on a single $1km^2$ map with 10,000 goals. Compared against a map specific LSTM baseline policy. Shown is goal distance, in 100m bins, compared with the success rate of each approach. We observe that the method begins to reach diminishing returns at 1,600 vertex locations. Overall average success rate is increased from 72% to 92%.

Distance Constrained Random Vertex Placement approach and evaluated the performance on 10,000 goals with graphs of size 200 to 1,600 nodes. We achieve an increase in success rate of 20% on all goals, shown in Figure 7.

Conclusions

We have introduced a set of graph augmented RL approaches for planning and navigation in vast 3D video game worlds. To demonstrate the capacity of these approaches we have introduced the GameRLand3D environment, an open-source reinforcement learning environment built in the Unity game engine. GameRLand3D enables procedural generation of vast open world environments with hazards such as water and lava, and features such as jump pads, buildings and overhangs. We have identified that end-to-end Deep RL approaches under-perform in large-scale environments. We evaluated various graph augmented RL techniques along three axis, trajectory based, random vertex placement and flood with grandchildren, providing a reasonable compromise between graph building time, run-time performance and success rate. Our hybrid approaches boost generalization performance from 69.6% to 86.2%. We also achieve a 20% increase in success rate in big environments.

In addition to support and maintenance of the GameRLand3D environment, our roadmap includes the addition of more complex building types, player abilities such as teleportation and jetpacks, and other features such as ravines and caves. Future hybrid approaches in this environment will investigate the possibility of online graph building, using function approximation to predict edge connectivity in dynamic environments where the graph must be reconstructed when there is a change in map topology.

References

Alonso, E.; Peter, M.; Goumard, D.; and Romoff, J. 2021. Deep Reinforcement Learning for Navigation in AAA Video Games. In Zhou, Z.-H., ed., *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, 2133–2139. International Joint Conferences on Artificial Intelligence Organization. Main Track.

Anderson, P.; Chang, A.; Chaplot, D. S.; Dosovitskiy, A.; Gupta, S.; Koltun, V.; Kosecka, J.; Malik, J.; Mottaghi, R.; Savva, M.; et al. 2018. On evaluation of embodied navigation agents. *arXiv preprint arXiv:1807.06757*.

Beattie, C.; Leibo, J. Z.; Teplyashin, D.; Ward, T.; Wainwright, M.; Küttler, H.; Lefrancq, A.; Green, S.; Valdés, V.; Sadik, A.; Schrittwieser, J.; Anderson, K.; York, S.; Cant, M.; Cain, A.; Bolton, A.; Gaffney, S.; King, H.; Hassabis, D.; Legg, S.; and Petersen, S. 2016. DeepMind Lab. *arxiv pre-print 1612.03801*.

Beeching, E.; Dibangoye, J.; Simonin, O.; and Wolf, C. 2020a. Learning to plan with uncertain topological maps. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part III 16*, 473–490. Springer.

Beeching, E.; Wolf, C.; Dibangoye, J.; and Simonin, O. 2020b. EgoMap: Projective mapping and structured egocentric memory for Deep RL. *arXiv preprint arXiv:2002.02286*.

Bentley, J. L. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9): 509–517.

Bhatti, S.; Desmaison, A.; Miksik, O.; Nardelli, N.; Siddharth, N.; and Torr, P. H. 2016. Playing Doom with slam-augmented deep reinforcement learning. *arXiv preprint arXiv:1612.00380*.

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. OpenAI Gym. *arxiv pre-print 1606.01540*.

Chaplot, D. S.; Salakhutdinov, R.; Gupta, A.; and Gupta, S. 2020. Neural Topological SLAM for Visual Navigation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

Chung, J.; Gulcehre, C.; Cho, K.; and Bengio, Y. 2015. Gated feedback recurrent neural networks. In *International conference on machine learning*, 2067–2075. PMLR.

Devlin, S.; Georgescu, R.; Momennejad, I.; Rzepecki, J.; Zuniga, E.; Costello, G.; Leroy, G.; Shaw, A.; and Hofmann, K. 2021. Navigation Turing Test (NTT): Learning to Evaluate Human-Like Navigation. In *2021 International Conference on Machine Learning*. Source code available at: https://github.com/microsoft/NTT.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1): 269–271.

Eysenbach, B.; Salakhutdinov, R. R.; and Levine, S. 2019. Search on the replay buffer: Bridging planning and reinforcement learning. In *Advances in Neural Information Processing Systems*.

Gordillo, C.; Bergdahl, J.; Tollmar, K.; and Gisslén, L. 2021. Improving Playtesting Coverage via Curiosity Driven Reinforcement Learning Agents. arXiv:2103.13798.

Graves, A.; Wayne, G.; and Danihelka, I. 2014. Neural turing machines. *arXiv preprint arXiv:1410.5401*.

Graves, A.; Wayne, G.; Reynolds, M.; Harley, T.; Danihelka, I.; Grabska-Barwińska, A.; Colmenarejo, S. G.; Grefenstette, E.; Ramalho, T.; Agapiou, J.; et al. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626): 471–476.

Gupta, S.; Davidson, J.; Levine, S.; Sukthankar, R.; and Malik, J. 2017. Cognitive mapping and planning for visual navigation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

Haarnoja, T.; Zhou, A.; Abbeel, P.; and Levine, S. 2018a. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*.

Haarnoja, T.; Zhou, A.; Hartikainen, K.; Tucker, G.; Ha, S.; Tan, J.; Kumar, V.; Zhu, H.; Gupta, A.; Abbeel, P.; et al. 2018b. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*.

Hart, P.; Nilsson, N.; and Raphael, B. 1968a. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968b. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2): 100–107.

Hochreiter, S.; and Schmidhuber, J. 1997. Long short-term memory. *Neural computation*, 9(8): 1735–1780.

Jaderberg, M.; Mnih, V.; Czarnecki, W. M.; Schaul, T.; Leibo, J. Z.; Silver, D.; and Kavukcuoglu, K. 2016. Reinforcement Learning with Unsupervised Auxiliary Tasks. *arXiv preprint arXiv:1611.05397*.

Johnson, M.; Hofmann, K.; Hutton, T.; and Microsoft, D. B. 2016. The Malmo Platform for Artificial Intelligence Experimentation. In *IJCAI*.

Juliani, A.; Berges, V.-P.; Vckay, E.; Gao, Y.; Henry, H.; Mattar, M.; and Lange, D. 2018. Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*.

Kapturowski, S.; Ostrovski, G.; Quan, J.; Munos, R.; and Dabney, W. 2018. Recurrent experience replay in distributed reinforcement learning. In *International conference on learning representations*.

Kolve, E.; Mottaghi, R.; Gordon, D.; Zhu, Y.; Gupta, A.; and Farhadi, A. 2017. AI2-THOR: An Interactive 3D Environment for Visual AI. *arxiv pre-print 1712.05474v1*.

Manolis Savva; Abhishek Kadian; Oleksandr Maksymets; Zhao, Y.; Wijmans, E.; Jain, B.; Straub, J.; Liu, J.; Koltun, V.; Malik, J.; Parikh, D.; and Batra, D. 2019. Habitat: A Platform for Embodied AI Research. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*.

Mirowski, P.; Grimes, M.; Malinowski, M.; Hermann, K. M.; Anderson, K.; Teplyashin, D.; Simonyan, K.; Zisserman, A.; Hadsell, R.; et al. 2018. Learning to navigate in cities without a map. *Advances in Neural Information Processing Systems*, 31: 2419–2430.

Mirowski, P.; Pascanu, R.; Viola, F.; Soyer, H.; Ballard, A. J.; Banino, A.; Denil, M.; Goroshin, R.; Sifre, L.; Kavukcuoglu, K.; Kumaran, D.; and Hadsell, R. 2016. Learning to navigate in complex environments. *arXiv preprint arXiv:1611.03673*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533.

Paine, T. L.; Gulcehre, C.; Shahriari, B.; Denil, M.; Hoffman, M.; Soyer, H.; Tanburn, R.; Kapturowski, S.; Rabinowitz, N.; Williams, D.; et al. 2019. Making efficient use of demonstrations to solve hard exploration problems. *arXiv preprint arXiv:1909.01387*.

Parisotto, E.; and Salakhutdinov, R. 2017. Neural map: Structured memory for deep reinforcement learning. *arXiv preprint arXiv:1702.08360*.

Perlin, K. 1985. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3): 287–296.

Savinov, N.; Dosovitskiy, A.; and Koltun, V. 2018. Semi-parametric topological memory for navigation. *arXiv preprint arXiv:1803.00653*.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of Go with deep neural networks and tree search. *nature*, 529(7587): 484–489.

Snook, G. 2000. Simplified 3D movement and pathfinding using navigation meshes. In *Game Programming Gems*, 288–304. Charles River Media.

van Toll, W. G.; Cook IV, A. F.; and Geraerts, R. 2012. A navigation mesh for dynamic environments. *Computer Animation and Virtual Worlds*, 23(6): 535–546.

Wani, S.; Patel, S.; Jain, U.; Chang, A.; and Savva, M. 2020. MultiON: Benchmarking Semantic Map Memory using Multi-Object Navigation. In Larochelle, H.; Ranzato, M.; Hadsell, R.; Balcan, M. F.; and Lin, H., eds., *Advances in Neural Information Processing Systems*, volume 33, 9700–9712. Curran Associates, Inc.

Wayne, G.; Hung, C.-C.; Amos, D.; Mirza, M.; Ahuja, A.; Grabska-Barwinska, A.; Rae, J.; Mirowski, P.; Leibo, J. Z.; Santoro, A.; et al. 2018. Unsupervised predictive memory in a goal-directed agent. *arXiv preprint arXiv:1803.10760*.

Wydmuch, M.; Kempka, M.; and Jaśkowski, W. 2018. ViZDoom Competitions: Playing Doom from Pixels. *arXiv preprint arXiv:1809.03470*.

Xia, F.; R. Zamir, A.; He, Z.-Y.; Sax, A.; Malik, J.; and Savarese, S. 2018. Gibson env: real-world perception for embodied agents. In *Computer Vision and Pattern Recognition (CVPR), 2018 IEEE Conference on*. IEEE.

Zhang, J.; Tai, L.; Boedecker, J.; Burgard, W.; and Liu, M. 2017. Neural Slam: Learning to explore with external memory. *arXiv preprint arXiv:1706.09520*.