Reinforcement Learning Agents with Generalizing Behavior

Reid Sawtell*, Anthony Chavez*, Sarah Kitchen*, Timothy Aris, and Chris McGroarty**

*Michigan Tech Research Institute (MTRI), 3600 Green Ct., Ann Arbor, MI 48103

** U.S. Army Combat Capabilities Development Command - Soldier Center (DEVCOM SC) Simulation and Training Technology Center (STTC), USA

*rwsawtel@mtu.edu, ajchavez@mtu.edu, snkitche@mtu.edu, **timothy.aris.civ@army.mil, christopher.mcgroarty.civ@army.mil

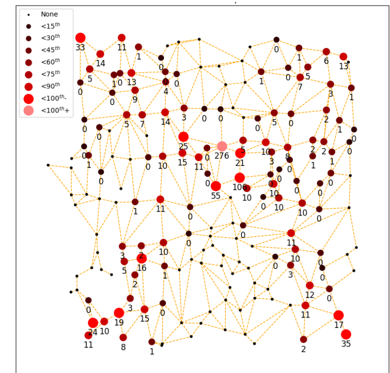
We explore the generality of Reinforcement Learning (RL) agents on unseen environment configurations by analyzing the behavior of an agent tasked with traversing a graph-based environment from a starting position to a goal position. We find that training on a single task is likely to result in inflexible policies that do not respond well to change. Instead, training on a wide variety of scenarios offers the best chance of developing a flexible policy, at the expense of increased training difficulty.

Figure 1: Example of a game environment with abstracted graph model overlain. The red and blue player each have an independently assigned random goal (orange, yellow) during training.

Our environment is formulated as a node/edge graph. The agent begins at a particular node on the graph and must traverse the graph to reach a goal node. The agent receives reward for occupying the goal node when the scenario terminates, and no reward otherwise. The scenario ends after a preset number of turns, or 3 turns after the agent first arrives at the goal node, whichever occurs first. The agent is controlled by a feed forward graph neural network trained using either Proximal Policy Optimization (PPO) (Schulman et al. 2017) or a Dueling Deep Q-Network (DQN) (Mnih et al. 2015) with prioritized replay buffer. The graph neural network produces output for each node of the graph, which is then trimmed to valid actions using an environment-provided action mask that limits the options to either remaining on the same node or transitioning to a neighbor node in the graph. The graph architecture includes a custom U-net (Ronneberger et al. 2015) style graph pooling and unpooling operation based on a hierarchical graph simplification (Sawtell, 2024). This allows information to rapidly propagate longer distances between nodes without recurrent operations or necessitating an overly deep network.

In our first experiment, we trained the agent using PPO on a scenario in which the goal is fixed at a node near the center of the graph. At the beginning of each scenario, the agent is randomly assigned one of four starting positions as the corners of the graph. The agent rapidly learns to converge on the central goal regardless of its starting position, and further reveals the agent can reach the central node from nearly any node regardless of it being one of the nodes in the training set. Unfortunately, if the goal position is moved, the agent behaves as if it were still in the central location: the agent learned a gradient to move towards a particular location on the graph but cannot generalize to changing that location.

Figure 2: Heatmap of visited nodes over 100 evaluations of the PPO derived policy. The highly visited central node is the goal location, while the start locations are in one of the four corners of the graphs.



Copyright © 2024 by the authors.
This open access article is published under the Creative Commons Attribution-NonCommercial 4.0 International License.

To attempt to overcome this failure, we modified the scenario such that both the start node and goal node are fully randomized within the graph, reasoning the increased scenario diversity would encourage a more general learned policy. We were not, however, successful in learning a performant policy, only succeeding if the agent began close to the goal node. This resulted in our switch from PPO to the off-policy DQN algorithm, which we reasoned might be more stable with the replay buffer to draw experiences from. The result is an agent that is capable of reaching the goal in approximately 88% of random scenarios. To further test the generality of the policy, we then tested the agent on a different graph without retraining. Performance dropped to success in ~23% of random scenarios. This indicates that through training, the DQN policy ("Actor network") is still memorizing something about the topology of the graph, despite the graph independence of the deep neural network architecture.

In order to define an AI agent in a simulation or game environment using the trained Actor network, we must define an interface that can translate abstract AI model to functions. To that end we used the Rapid Integration and Development Environment (RIDE), a platform designed for prototyping synthetic training scenarios, features, and AI agents, built on top of the Unity game engine (Hartholt, *et al*., 2021). To control character behaviors within Unity, it is essential to continuously observe and interpret the environment. These observations are then passed into the AI Actor network, which, in turn, dictates the movements of agents within the game or simulation. Achieving this level of interaction is made possible through the use of the Unity Machine Learning Agents Toolkit (ML-Agents Toolkit) (Lanham, 2018). ML-Agents serves as a bridge, connecting Unity C# scripts with Python scripts, thus facilitating communication through a gym interface. This setup allows for an easy way to exchange data and instructions between the game engine and the AI model.

The key to managing the complexity of a scenario with dozens or more Agents is a custom component known as the "board". This element of the interface architecture is tasked with maintaining the state of the environment in Unity and is initialized with comprehensive scenario details. This includes the locations of graph nodes as well as agent spawning details, providing a knowledge base that can be abstracted for agent integration. In addition to initialization, the board actively relays movement updates to agents and monitors their positions, which are instrumental for defining observations. This mechanism ensures a detailed and up-to-date model of the environment is available to the agent for each action request.

The Unity C# script uses the "board" to capture the current state of the environment at each simulation step and sends the data to the Python script. Following this, the AI Actor processes these observations and determines the appropriate actions to be taken by the agents. These actions are represented as an array of integers, where each integer corresponds to a specific node identifier within the environment. Importantly, each position within the action array is linked to a distinct team, enabling the "board" to give agents their actions.

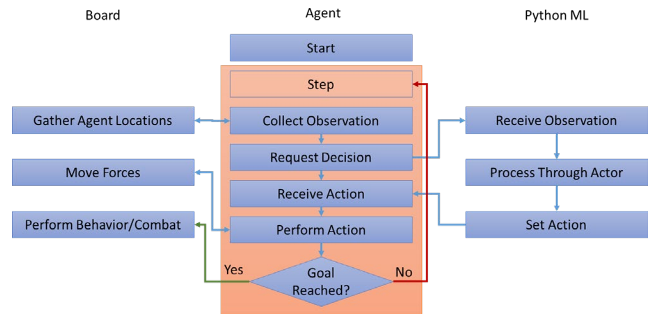


Figure 3: The scenario data can be changed via the board, and the Actor network can be changed independently, without changing the Agent infrastructure.

Once an agent, under the guidance of AI, reaches its designated goal, there's a shift in its behavior tree. This transition moves from actively seeking the next action to engaging in predefined local behaviors, such as patrol routines. Specifically, a patrol behavior may direct an agent to follow a designated route around its current location, offering a more lifelike and engaged action compared to merely standing idle at the goal. In the context of our scenario, this dynamic plays out as such: the first team to secure the goal node initiates a patrol behavior, diligently monitoring their surroundings. This state persists until they encounter the opposing team within their vicinity, at which point, a strategic confrontation, or 'firefight', unfolds. This implementation of local behaviors facilitates a smooth transition of control between the AI's strategic movement commands and the agent's autonomous actions, as well as introducing a layer of complexity within the environment.

Acknowledgements

This material is based upon work supported by the US Army Combat Capabilities Development Command Soldier Center (DEVCOM SC) Simulation and Training Technology Center (STTC) under contract No. W912CG-21-C-0016. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the DEVCOM SC STTC.

References

Hartholt, A., McCullough, K., Fast, E., Leeds, A., Mozgai, S., Aris, T., ... & McGroarty, C. (2021). Rapid prototyping for simulation and training with the Rapid Integration & Development Environment (RIDE). In Proceedings of the 2021 Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC).

Lanham, M. (2018). Learn Unity ML-Agents–Fundamentals of Unity Machine Learning: Incorporate new powerful ML algorithms such as Deep Reinforcement Learning for games. Packt Publishing Ltd.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). Human-level control through deep reinforcement learning. nature, 518(7540), 529-533.

Ronneberger, O., Fischer, P., & Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. In Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18 (pp. 234-241). Springer International Publishing.

Sawtell, R., Kitchen, S., McGroarty, C., and Aris, T. 2024. Learning Cohesive Behaviors Across Scales for Competitive Agents. In *The International FLAIRS Conference Proceedings*. Vol. 37.

Schulman J., Wolski F.,Dhariwal F., Radford A., and Klimov O. 2017. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347, 2017.