Learning Cohesive Behaviors Across Scales for Semi-Cooperative Agents

Reid Sawtell*, 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, snkitche@mtu.edu, **timothy.aris.civ@army.mil, christopher.mcgroarty.civ@army.mil

Abstract

The development of automated opponents in video games has been part of game development since the very beginning of the field. The advent of modern AI approaches, such as reinforcement learning, has opened the door to a wide variety of flexible and adaptive AI opponents. However, challenges in producing realistic opponents persist, namely scalability and generalizability. Scalability is of particular importance when many individual opponents are required to act cohesively over long distances, but this makes learning more difficult. This paper presents a novel architecture applying graph convolutional layers in a U-net with custom pooling operators in order to achieve learning across scales. League play reinforcement learning was used to train competitive agents in a navigation mesh environment.

Introduction

Reinforcement learning (RL) provides an optimal policy approximation approach for a wide variety of complex problems, including game strategy development. Actor-critic methods in particular can be formulated as Counterfactual Regret Minimization (CFR) methods in Multi-Agent RL (MARL) settings, and therefore converge to a Nash equilibrium when the MARL problem can be formulated as a zero-sum extensive form game (Lockhart, et al. 2019). General-sum games are more complex, as they consist of both fully cooperative (potential game), harmonic, and non-strategic components (Candogan, et al. 2011). The setting of AI agent development for combat training simulations, and combat more generally, regularly violates the zero-sum assumption. Therefore, general sum games, or semi-cooperative stochastic game models, should be considered (Kitchen, et al. 2023).

Development of scalable computational models and approaches for such models is a challenging problem. The approach taken in this paper is a MARL approach using Deep Reinforcement Learning (DRL) in an actor-critic framework for a pair of agents. The state is fully observable and the environment is deterministic, but the rewards are private for each agent, and only semi-cooperative, meaning joint actions of the agents can negatively impact both agents. Simulation play training provides an approach for developing best response policies for multiple agents, either sequentially or jointly, though effective implementations are an open and active area of research, including research into state and action representations, scalable architectures, mitigation and impact of non-stationarity, and associated pathologies (Huh and Mohapatra, 2023). This paper presents a novel neural network architecture incorporating a hierarchical graph model for an environment and associated experiments with the trained AI agents. Experimental results show behaviors consistent with strategies of general-sum games.

Methods

Gameplay Environment

Our game environment is derived from the navigation mesh in a Unity-based simulator. Any undirected graph $G = (V, E)$ can serve as the game environment, where the nodes V represent terrain patches and the edges E are adjacencies between them. The game is a dynamic battlefield game in which a blue (B) and red (R) team act simultaneously to maneuver in the environment and capture objective terrain. Figures 3-5 show an abstract graph environment obtained through a node-aggregation method applied to a navigation mesh from an environment implemented in the Unity game engine.

Each player starts with 10 squads composed of multiple sub-units. Let $\boldsymbol{b} = (b_v)_{v \in V}$ denote the vector of squad distributed over the graph for the blue team, and correspondingly define $\boldsymbol{r}$ for the red team. On each turn, both players specify the actions each squad will take, either transitioning to a neighboring node or remaining in place. Movement is resolved concurrently for both players. If after resolving movement, two or more squads from both players occupy the same node, combat occurs. Combat follows a Lanchester model of attrition
$b_v[t+1] = b_v[t] - \alpha r_v[t],\ \ r_v[t+1] = r_v[t] \ - \beta b_v[t]$
in which the sum of sub-units of both players at a node $\boldsymbol{b}[t], \boldsymbol{r}[t]$ at time t determines the casualties the opponent will take, which are then randomly removed from the squads present at the battlefield node (Taylor, 1979). The scalar multipliers α and β are defined by node and are used as combat modifiers representing fortification of the position at that node. For these nodes, the first player that occupies the node for more than one turn receives a bonus multiplier to the attrition parameter. This bonus will begin the turn after the node is occupied and gradually increases until reaching a maximum after the third turn after the node is occupied. The bonus is lost if the player no longer occupies the node, but otherwise squads can come and go freely without losing the bonus so long as at least one remains at the end of any turn. Abandoned nodes reset to their original combat multiplier value and may be re-occupied by any player with the same effect.

Our environment is highly configurable, allowing it to be used to define and autonomously play a wide variety of scenarios on arbitrary graphs. This is accomplished by means of a scenario initialization function, configuring the initial environment parameters and goal weights $\boldsymbol{g} = (g_v)_{v \in V}$ for each player, subject to the constraint such that $\sum_{v \in V} g_v = 1$. The functionalization of this process is needed to specify complex configurations, such as randomized start or goal locations in the reinforcement learning process described below.

Objectives and Rewards

Since our game is based on battlefield style games, the objective for each player is to capture terrain specified in initialization. A terminal reward for a game is defined by a distribution of forces over one or more goal nodes with a priority weight that indicates the percentage of forces which should ideally occupy that node. Maximum reward is achieved if the player occupies every goal node with the correct proportion of forces when the scenario ends. A node is considered occupied if the player has at least one squad with units remaining and no opponent units are alive at that node. Partial rewards are achievable, with each occupied goal node contributing based on its priority.

To account for the distribution, the goal reward is modulated by the Earth Movers Distance (EMD) needed to achieve perfect distribution over all objectives, as well as an attrition penalty for losing too many units in combat. The EMD between two discrete distributions on a weighted graph is the normalized work required to flow one distribution to the other with a cost-minimizing flow. In our model, each team's forces are converted to a distribution over the environment graph via normalization, so the EMD between the current position of a player's forces and the goal distribution for that team effectively calculates the shortest path distance to reach the goal, in the absence of any combat. For simplicity, we show only the reward function definitions for the blue team, with $\hat{\boldsymbol{b}}$ denoting the normalized force vector. The terminal reward for B is defined by

$$R_B = \frac{L(\boldsymbol{b}) R_{goal}}{1 + \gamma EMD\left(\hat{\boldsymbol{b}}, \boldsymbol{g}\right)}$$

where L is an attrition penalty that is < 1 if too many forces are lost, γ is a weight, and

$$R_{goal} = \frac{\sum_{v \in V} \chi_{b>0}\left(\hat{b}_v\right) \sqrt{g_v}}{\sum_{v \in V} \sqrt{g_v}}$$

where χ denotes an indicator function. In the development of these reward functions, we chose to prioritize the relationships between the units and the objective terrain goals. They are rewarded for capturing their goals, but the inclusion of EMD provides a metric for the players to be rewarded for positioning forces near their goals in a way that is meaningful in a distributional sense. By also including a penalty for losing too many forces via L, the player can learn to "fall back" to nearby positions to their goals if the losses for capturing the objective terrain are too high.

To encourage movement towards the goals, an incremental reward also based on the EMD is employed. This reward is needed for the network to begin to converge on a solution on larger graphs but is not required for smaller graphs. It is deliberately smaller than the achievable goal rewards in order to not unduly bias the agents towards shortest path movement at the expense of strategy. A common issue in reinforcement learning is defining the right balance of incremental and sparse rewards, particularly when the horizon to the terminal condition in a training episode is long. Alternative incremental rewards, such as those including a time decay parameter, or related curriculum training strategies, could be used to improve the policies derived in this model in the future. The incremental reward at time t is defined by

$$R_{inc}[t] = E_{diff}/D_{diff}[t] + E_{max}/D_{max}[t]$$
$$D_{diff}[t] = 1 + \gamma_{inc} EMD(\hat{\boldsymbol{b}}[t], \boldsymbol{g})$$
$$D_{max}[t] = D_{diff}[t] - \gamma_{inc} \max_{\tau<t}\{EMD(\hat{\boldsymbol{b}}[\tau], \boldsymbol{g})\}$$

where E_{diff} and E_{max} are constants, and γ_{inc} is a weight. In the implementation used for experimentation, γ_{inc} was an order of magnitude larger than γ.

Agent Actions and Observations

Each player in the game is controlled by a neural network agent, consisting of a feed forward graph neural network. The network takes as input the observations generated by the environment and outputs agent controls. Observations are represented as node attributes and include the per squad location, the proportion of player forces, the proportion of opponent forces, the goal weight, player fortification bonus, and opponent fortification bonus. The player and opponent force proportions are normalized relative to the maximum number of units any player has accumulated on any one node of the graph. Notably, the coordinates of the nodes are not included. The output consists of a per-squad per node weight. The environment also provides an action mask which is used to eliminate impossible actions. The remaining values are used in a standard softmax probability distribution to select the node transition.

Additionally, the network outputs a separate per squad "sticky" action probability. The sticky action also follows a softmax probability approach to determine if it is enabled or not for a given squad. The result is that all squads that occupy the same node, and have the sticky action defined, will act in concert. This is achieved by randomly choosing one of the actions of the participating squads and changing all of those squads' actions to that action. This additional specification is necessary to allow the AI agent to more deliberately control multiple squads by effectively aggregating them into a single larger unit via the sticky action. Without it, the agent can only probabilistically assign actions to each individual squad, which provided the course of action is not crystal clear, allows for random walking actions according to the current policy. Consider a simple scenario in which all squads occupy the same node, and there are two neighbor nodes which are of equal interest to visit, but the agent wishes to keep its forces massed. Specifying 50% squad transition probability to each neighbor node will result in roughly half of the squads at each node, when what the agent really wants is to specify that all squads will go to one of the two nodes with 50% probability. With the sticky flag specified, the agent is able to group the squads, ensuring that the group will move together to one of the two nodes.

Network Architecture

Our network architecture is a graph convolutional U-Net (Ronneberger et al. 2015) with skip connections, implemented in PyTorch. The U-Net architecture is necessary to transmit information across many nodes in the graph. Without it, the agent can only learn on small graphs or scenarios in which it starts fairly close to its goal nodes. Pooling and unpooling across layers of the U-net is implemented using a custom hierarchical mesh aggregation scheme. The initial graph, which is a graph covering the topology of the environment, is fed through a quadric mesh simplification algorithm, which reduces the size of the mesh using edge collapse. Some advantages of this algorithm are that it does not require a planar graph, and edge collapse preserves the connectivity of the reduced graph relative to the original. When the mesh size reaches certain milestones (4, 16, 64, and 256x reduction), the reduced graph is saved as well as the mapping from the previous graph to the new one. In the implementation used for the experiments, five graphs, as well as the four mappings between them, are inputs to the network module. Pooling and unpooling are accomplished with matrix multiplication: Node features across nodes contracted to the same node in a smaller graph are added when pooling into the reduced graphs, and node features are replicated across nodes in the preimage of a coarse node when unpooling to the finer graphs.

We found graph convolution by itself (PyTorch GCNConv) to be insufficient for the network to learn. As an alternative, we altered the formulation by adding skip connections around each GCNConv layer followed by a linear layer, as shown in Figure 1.

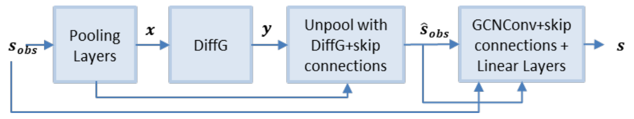
s_{obs}
Pooling Layers
x
DiffG
y
Unpool with DiffG+skip connections
$\hat{s}_{obs}$
GCNConv+skip connections + Linear Layers
s

Figure 1 Architecture overview for backbone network used in both actor and critic networks.

This block allows a network to encode the relative impact of diffusion across the graph obtained by a single message passing step in the GCNConv. A similar network block incorporating skip connections from the pooling layers is applied between unpooling layers. An overview of the architecture is shown in Figure 2.

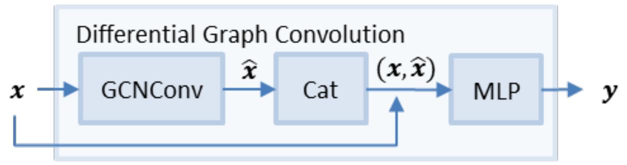

Figure 2 Architecture of a Differential Graph Convolution (DiffG) block.

This forms the backbone network architecture and is used in both the actor and critic, though weights are not shared. The output of each network (actor and critic) is obtained by passing the representation $\boldsymbol{s}$ into linear layers of appropriate dimensions. This backbone successfully achieved learning whereas alternative strategies, such as Diffpool aggregation layers or deeper GCNConv network architectures, could not. Nonetheless, the networks will still fail to learn without the addition of the continuous EMD-based rewards, which we had avoided adding initially on the assumption that it would bias the agent too much towards shortest path behavior, and prohibit more complex strategies, such as taking a longer path to avoid a heavily fortified node.

Training

Agents are trained using a custom distributed PPO actor critic formulation (Schulman et al. 2017). Since the game is two player, separate actor and critic networks are initialized for each player resulting in a total of four networks. A number of client processes are started, which can be on the host computer or any networked computers. The client processes each spin up a specified number of child workers, which are responsible for running the environment simulation and returning the experiences. The "head" process is responsible for specifying the particular graph and game scenario that will play out in the client environments. At the start of each epoch, the head process shares the current network weights and environment graph and initializer function with all connected client processes. The number of required environment steps is then split into a configurable number of jobs which are placed on a queue for the client processes to pick up. The client processes then simulate the required steps and return all relevant information, such as diagnostic information, the action history, advantages, loss, etc. The head process waits for all client jobs to return, then aggregates the returned values into an experience buffer, which is passed to the PPO algorithm to generate the network loss. Then, an Adam optimizer updates the network weights. Since each player acts concurrently, and has distinct state space observations, it is viable to train both networks at once. Training data from each simulation is gathered for both players, and the updates are performed separately for each player. For our PPO algorithm, we specify a minibatch size of 250 steps, with 4 mini-batches per epoch (1000 environment steps per epoch). The PPO update terminates after 10 iterations or KL divergence exceeds a value of 0.01. We did not use entropy loss for the experiments presented here, but we did include it as a capability in our implementation. The Adam optimizers were configured with a learning rate of 1e-4.

Our setup allows for curriculum training since the scenarios can be altered at each epoch, and we have found that limiting the scenario length initially can accelerate convergence in learning to navigate squads towards the goal nodes. However, this was unnecessary for the scenarios presented here.

Experiments

We used a single graph hierarchy to perform two experiments with our environment and network architecture. The same graph was used for all experiments and was processed through the hierarchical graph simplification procedure to produce the graph and mapping set necessary for the graph pooling algorithm. Each network was allowed to train for a period long enough that the behavior was no longer significantly changing, which typically takes between 2000-12000 epochs depending on scenario complexity. All experiments were trained on a Dell Precision workstation with a Xeon 8 core CPU and NVIDIA GeForce RTX 3080 laptop GPU. The client was configured with 7 parallel environment threads. Additional networked clients were not used for these experiments. A single epoch averages around 7 seconds, leading to total training times of 4-24 hours. The majority of the training time, approximately 4/5, was spent simulating the environment steps. The environment steps are typically slower since the networks are evaluated on CPU since they cannot be batched.

Experiment 1: Impact of Relative Force Strength on Learned Policies

Red starts with all squads in the upper left corner of the map, while blue starts in the lower left. Both players have the same goal node near the center of the map, which is 12 steps away from either starting position. The goal node is the only node that can be fortified. Two sets of policies were trained: one set with Blue initialized with 1/10th the forces of Red, putting it at a distinct disadvantage (i.e. the "weak" Blue player); the other with equally-matched forces (i.e. the "strong" Blue player). The question we aimed to answer with this experiment was how the influence of EMD in the incremental reward would be impacted as the combat power of the opposing forces increased from unbalanced to equally balanced. Since the EMD incentivizes each AI player to route its forces to the goal node along the shortest path in the graph, one expected impact of increasing attrition by changing the force numbers would be less direct trajectories to a goal to avoid the opposing force.

Experiment 2: Impact of Distributed, Semi-Cooperative Goals on Learned Policies

In this experiment, Red randomly starts in one of 4 corner positions while Blue spawns in the opposite position to that which Red spawns in. The central primary goal node remains the same for both players, but each has a collection of 6 secondary goal nodes in close proximity to the primary goal node. Three of these secondary goal nodes are shared for each player, while the other three are distinct. The goal nodes do not change from run to run, only the start positions. Both players have evenly matched forces. In this experiment, the primary goal node cannot be fortified but its neighboring nodes, which includes some of the secondary goal nodes, can be. The objective of this experiment was to determine the impact of the semi-cooperative goals on the learned policies. Namely, whether attrition to decisively capture the shared goals was preferred to the non-conflict goals. Fortification in the non-conflict nodes should allow each player to adversely influence the trajectory of the opposing player towards the common goal.

Results

In Experiment 1, the weak Blue player learns a shortest path from its start position to the goal while red learns a path that is one step longer than the shortest, and also displays more diffusive behavior in its learned policy. This is a consequence of the force imbalance between the weak Blue and Red forces, since Blue needs its full mass to achieve measurable attrition against Red, but Red does not need its full force to achieve overmatch vs. the weak Blue. Compared to the equally matched strong Blue vs. Red, both players demonstrate expected trajectories that are mostly directed along the shortest path, with some diffusivity. Both players must effectively mass forces at the goal to have any hope at capturing it (Figure 3).

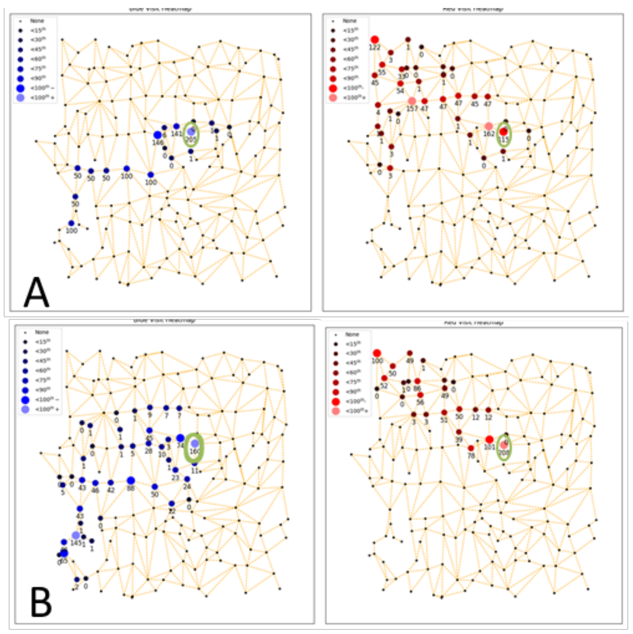

Figure 5 Visitation rates of learned policies for (A) The weak Blue vs. Red policies and (B) the strong Blue vs. Red policies. The goal node is circled in green.

Notably, in the learned policies for the weak Blue player, the Red and Blue agents alternate positions between the goal node and adjacent node. While the Red player lands on the goal node as the scenario ends the majority of the time, Blue exploits mis-timings by Red and claims the goal for itself. Combat is relatively rare, likely due to occasional blue/red getting out of step in the dance. Attrition of the opposing force is not incorporated into the reward structure, so this is an emergent cooperative behavior, reflecting the expectation that the policies should converge to an equilibrium that jointly optimizes the expected reward for both players. Since Red can always capture the goal, the learned policy balances the ability to capture the goal in expectation while minimizing force loss. Combat at the goal node is instead required when the forces are equally-matched.

The policies learned in Experiment 2 demonstrate an uneasy truce between Red and Blue (Figure 4). Blue pursues the primary goal and a few of its secondary goals. Red settles for several of its secondary goals but places much of its forces on one of the sub-goals nearest the primary goal, which helps reduce the EMD penalty to not occupying the primary goal. Combat is fairly rare and happens mostly when Blue passes through Red to get to the primary goal.

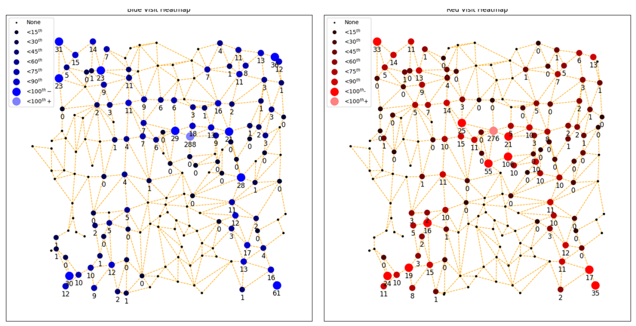
Figure 4 Visitation rates of learned policies for policies trained with random opposite starting corners and semi-cooperative goals

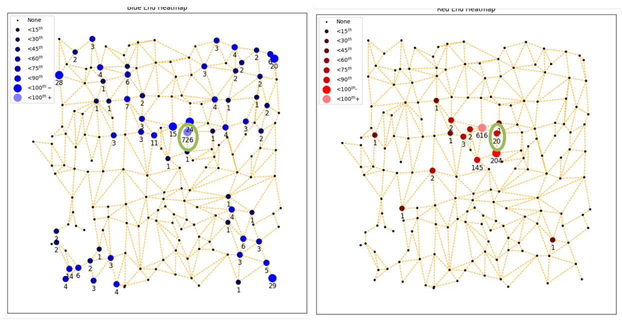
Figure 3 Visitation rates for terminal states in Experiment 2. Blue tends to capture the goal node, with Red populating adjacent sub-goals.

These experiments show that a relatively small graph convolutional network, with a few modifications, can learn interesting behaviors in a combative graph environment. In our testing, it was critical to include our custom graph pooling, the differential convolution block, as well as the continuous EMD reward to obtain policies that converged via training. Lacking any of these, the policies fail to converge to cohesive behaviors and either demonstrate diffusive behavior or to taking no action.

To further understand what the networks have learned, we tested the trained actor networks from Experiment 2 by changing scenario conditions, such as the start and goal locations. The policy behavior showed almost no response to a change in the goal or initial position, implying therefore that the location of the goal used in training was encoded into the learned weights of the network.

To further investigate this, we probed the network response to deliberate changes in the input observation. Changing the goal position has a slight effect on the probability distributions (Figure 6, top), but not enough to change the overarching behavior. A similar lack of impact is noted for changing the Blue force location (Figure 6, bottom). Changing the node fortification bonuses for the map has a considerable effect that results in diffusive behavior (Figure 6, middle). We hypothesize that the actor network has learned to treat the fortification bonuses as topological features in the graph, and this plays a significant role in the exhibited trajectories for the trained agents.

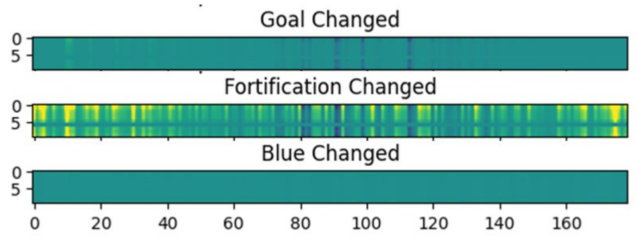

Figure 6 Activation changes in Red actor network from Experiment 2 subject to state input changes (Goal Location, Fortification, and Blue Position).

Conclusions

The implemented architecture demonstrates successful MARL training for a large graph environment and incorporates learned force aggregation behaviors via the “sticky” action. However, the trained policies in this model are “conflict-avoidant” since the reward function only incorporates penalties for attrition of one’s own forces and no rewards for attrition of the opposing force. The joint training produces policies that are emergently cooperative, despite the rewards being only semi-cooperative.

The impact of the hierarchical graph architecture is that our actor networks learned an approximately shortest path to a desired node, with a strong indication that the location of the goal nodes is encoded in the learned network weights. However, the semi-cooperative reward structure for the agents resulted in policies that behave as strategies in general sum games. Namely, although the reward structures are private to the individual game agents, cooperative joint strategies are demonstrated by the trained policies, such as the weak Blue learning to “dance” with Red in Experiment 1 in the hopes of exploiting a flaw in the timing of the Red agent ending the scenario on the objective node. Attrition is minimized for both teams, though it is only significantly costly for Blue when overmatched by Red. Similarly, in Experiment 2, Red and Blue converged to the cooperative strategy in which Blue captured the conflicting goal node, and Red learned to be satisfied with nearby sub-goals, thereby avoiding costly combat with an opponent.

By performing introspection on the actor network, we verified that the learned weights of the network are encoding the node locations and topological information, including a topological representation of the location of fortified nodes, which substantially change the ability of forces to maneuver through the environment.

Recommendations

Achieving better generalization of performance will be a focus for further research. One possible avenue is to allow networks to observe a wider variety of scenarios, much like Experiment 2 where the Red team learned to get to its desired goal from nearly any position in the map, even if it rarely, if ever, visited those nodes during training. Additional investigation into network capacity may be required in order to expand the training conditions, as our networks are relatively small to promote faster training times.

An alternative approach worthy of exploration is the decomposition of the combat problem into maneuver and combat sub-games, in which the incremental rewards in this paper are over-weighted in the maneuver game, which would be used to flow the forces into a “combat zone”, and a smaller graph neighborhood is used for the combat sub-game using only the sparse terminal reward. In order to combine the maneuver and combat sub-games, an options framework could be incorporated to switch between the two games (Vezhnevets, et al. 2019).

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

Candogan, O., Menache, I., Ozdaglar, A. and Parrilo, P.A., 2011. Flows and decompositions of games: Harmonic and potential games. *Mathematics of Operations Research,* 36(3), pp.474-503.

Claus, C. and Boutilier, C., 1998. *The dynamics of reinforcement learning in cooperative multiagent systems.* AAAI/IAAI, 1998(746-752), p.2.

Huh, D. and Mohapatra, P., 2023. Multi-agent Reinforcement Learning: A Comprehensive Survey. arXiv preprint arXiv:2312.10256.

Kitchen, S. and Brawner, K., 2022, May. Aggregation of Hierarchically-Organized Agents in a Multi-Agent System. In *The International FLAIRS Conference Proceedings* (Vol. 35).

Kitchen, S., McGroarty, C. and Aris, T., 2023, May. Model Representation Considerations for Artificial Intelligence Opponent Development in Combat Games. In *The International FLAIRS Conference Proceedings* (Vol. 36).

Lockhart, E., Lanctot, M., Pérolat, J., Lespiau, J.B., Morrill, D., Timbers, F. and Tuyls, K., 2019. Computing approximate equilibria in sequential adversarial games by exploitability descent. arXiv preprint arXiv:1903.05614.

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.

Schulman J., Wolski F.,Dhariwal F., Radford A., and Klimov O.

2017. Proximal policy optimization algorithms. arXiv preprint

arXiv:1707.06347, 2017.

Taylor, J. G. (1979). Attrition modelling. Operationsanalytische Spiele für die Verteidigung, 139-89.

Zhang, K., Yang, Z. and Başar, T., 2021. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *Handbook of reinforcement learning and control*, pp.321-384.

Vezhnevets, A. S., Wu, Y., Leblond, R., & Leibo, J.Z., 2019, Options as responses: Grounding behavioural hierarchics in multi-agent RL. Retrieved from arXiv preprint: arXiv:1906.01470.