GraphChase: A Platform and Benchmark for Urban Network Security Games

Shuxin Zhuang*
City University of Hong Kong
CAIR, Hong Kong Institute of Science & Innovation
Hong Kong, China
shuxin.zhuang@my.cityu.edu.hk

Shuxin Li*
Nanyang Technological University
Singapore, Singapore
shuxin.li@ntu.edu.sg

Tianji Yang
Georgia Institute of Technology
Atlanta, Georgia, USA
tyang425@gatech.edu

Muheng Li
University of Toronto
Toronto, Canada
muheng.li@mail.utoronto.ca

Xianjie Shi
The University of Hong Kong
Hong Kong, China
xianjieshi@connect.hku.hk

Bo An
Nanyang Technological University
Singapore, Singapore
boan@ntu.edu.sg

Youzhi Zhang†
CAIR, Hong Kong Institute of Science & Innovation
Hong Kong, China
youzhi.zhang@cair.cas.org.hk

Abstract

After the achievement of solving two-player zero-sum games, more AI researchers focus on solving multiplayer games. Urban Network Security Games (**UNSGs**) represent a class of such games, modeling real-world scenarios where law enforcement must strategically allocate limited resources to intercept criminals escaping within urban networks, and have gained considerable research attention. However, progress in this field has been limited by the absence of a standardized experimental platform and realistic benchmarks with heterogeneous travel costs. To address this limitation, we introduce **GraphChase**, an open-source platform designed to support the development and evaluation of algorithms for UNSGs. GraphChase offers a unified environment for modeling diverse UNSG variants on unweighted and weighted road networks across urban topologies. It also incorporates learning-based algorithms as baseline references for researchers. Furthermore, our experiments with GraphChase reveal that existing approaches to UNSGs still face challenges in terms of robustness and scalability, and suffer performance degradation when deployed under weighted edge costs, highlighting a sim-to-real generalization gap. GraphChase thus provides a realistic testbed for developing and validating UNSGs solvers under realistic travel-time heterogeneity.

CCS Concepts

• **Theory of computation** → *Multi-agent learning*; *Network games*; *Reinforcement learning*.

Keywords

security games, multiplayer games

ACM Reference Format:
Shuxin Zhuang, Shuxin Li, Tianji Yang, Muheng Li, Xianjie Shi, Bo An, and Youzhi Zhang. 2026. GraphChase: A Platform and Benchmark for Urban Network Security Games. In *Proceedings of KDD 2026 Datasets & Benchmarks Track (KDD '26).* ACM, New York, NY, USA, 17 pages.

*These authors contributed equally to this work.
†Corresponding author.

1 Introduction

In the field of AI research, a lot of focus has been placed on computing Nash equilibria [39, 46] in two-player zero-sum extensive-form games, where the players receive opposing payoffs [10, 38, 66]. In this scenario, a Nash equilibrium can be computed in polynomial time with respect to the size of the extensive-form game [46]. Recent significant achievements, such as achieving superhuman performance in the heads-up no-limit Texas hold'em poker game [10, 38], demonstrate that researchers have a strong theoretical and practical understanding of how to compute Nash equilibria in two-player zero-sum extensive-form games. However, computing Nash equilibria in multiplayer games remains an open challenge and is more difficult than in the two-player zero-sum setting [14, 62]. Consequently, more and more AI researchers have turned their attention to solving multiplayer games [11, 12, 16, 36, 56, 64].

As a specific instance of a multiplayer game, urban security games have earned considerable attention from researchers due to their practical significance [32–34, 52, 53]. In urban environments, ensuring public safety and security is crucial for law enforcement agencies. One important issue is the substantial number of innocent bystanders injured or killed during police pursuits [44]. It is therefore essential to develop effective strategies that enable multiple officers to apprehend fleeing criminals while minimizing risks to civilians and property damage. This paper focuses on responding to major incidents such as terrorist attacks or bank robberies, where police officers must swiftly intercept attackers during their escape. Such scenarios demand efficient strategies for criminal apprehension, which can be naturally modeled as UNSGs.

However, solving UNSGs is NP-hard [27, 63, 65]. More specifically, the strategy spaces of players in UNSGs are too large to be enumerated due to the memory constraint [27, 65]. Moreover, when players lack real-time information, they must make decisions under imperfect information. In addition, if police officers are unable to communicate during gameplay, they have to make decisions independently. Finally, UNSGs simultaneously involve cooperation among police officers and competition between the criminal and the team of police officers. To solve UNSGs, prior works have developed

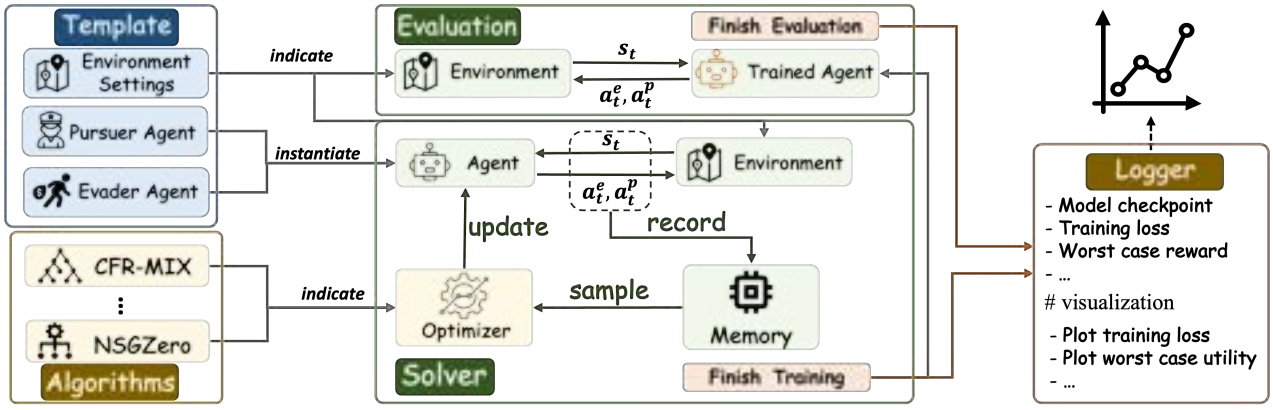

Figure 1: The blueprint of our GraphChase platform.

a range of sophisticated algorithms. Specifically, Counterfactual Regret Minimization (CFR) [66] has been extended to CFR-MIX [34] with deep learning enhancements [9]. Additionally, Neural Fictitious Self-Play (NFSP) [20] has been adapted into NSG-NFSP [53] and NSGZero [52]. Furthermore, the Policy-Space Response Oracles (PSRO) framework [31] has also been advanced into Pre-trained PSRO [33] and Grasper [32] to improve generalization. All of them are based on the state-of-the-art (SOTA) game-theoretical learning frameworks, powered by deep function approximation.

Although these efforts have achieved some progress in specific problem settings, the UNSGs research community still faces the following challenges. First, *the absence of a unified interface* results in incompatible implementations. Custom environments often return inconsistent data types, data structures, and action spaces. Consequently, an algorithm trained in one environment is often incompatible with others, making cross-evaluation and fair comparison difficult. Second, there is a *realism-scalability trade-off* in environment modeling. While optimization-based methods [63] incorporate edge weights to model travel times, they rely on mixed-integer linear programming (MILP) and thus struggle to scale to large-scale networks due to computational complexity. Conversely, SOTA learning-based methods typically simplify the environment to unweighted graphs (or uniform costs) to facilitate training. This simplification overlooks the real-world factor: the heterogeneity of road segments (e.g., varying lengths and speed limits) can alter strategies. Neglecting these realistic weights limits the effectiveness of algorithms in real-world scenarios.

To address these challenges, we introduce an open-source platform, **GraphChase**, to standardize UNSGs research and enable simulations in realistic environments. GraphChase is designed to provide researchers with a unified framework, as illustrated in Figure 1, facilitating both the development and comparison of algorithms. Specifically, our main contributions are as follows:

1) Development of a Unified UNSGs Platform. GraphChase provides a standardized interface that decouples the environment from the algorithm. This unification resolves the incompatibility issue, enabling cross-evaluation of different strategies on identical tasks. The platform allows researchers to define custom graph structures and configure key parameters, such as the number and positions of pursuers and evaders. It also includes optimized environment rollout mechanisms for efficient data generation.

2) Establishment of Benchmark Results. GraphChase integrates several deep learning-based algorithms within a unified game framework, enabling fair comparisons across diverse game configurations. The resulting benchmarks offer reliable baselines for evaluating future algorithms.

3) Revealing the Sim-to-Real Gap. We conduct extensive experiments on the platform to evaluate benchmark algorithms. The results reveal that current approaches face limitations in scalability and robustness, and suffer performance degradation when deployed on weighted road networks. These findings quantify a sim-to-real generalization gap and establish reference results for future work.

GraphChase lowers the entry barrier to UNSGs research and fosters a community focused on scalable, realistic security challenges.

2 Urban Network Security Games

We firstly define urban network security games (UNSGs) to model the interactions between multiple pursuers (police officers) and evaders (criminals) in the urban road networks. Then, building upon prior work on security games in urban environments [27, 63, 65], we highlight the core challenges involved in solving these games.

2.1 Game Definition

Consider the scenario in which pursuers are tasked with capturing evaders escaping through an urban road network. To model such interactions, we introduce the concept of UNSGs. Urban roads and pathways can be naturally represented as graphs, where intersections correspond to nodes and streets to edges. Formally, the graph is defined as $G = (V, E)$, where V denotes the set of vertices, and E denotes the set of edges. In graph G, we define a subset of the vertices, $V_{\text{exit}} \subset V$, as exit nodes from which the evader can escape, and let $\mathcal{N}(v)$ denote the set of neighbors of vertex v. In UNSGs, graphs may be directed or undirected to represent one-way and two-way streets, and weighted or unweighted, where edge weights capture differences in travel cost or terrain. This graphical representation provides a flexible way to simulate the game setting.

Without loss of generality, we assume the game is played on a weighted graph $G = (V, E, \omega)$, where $\omega(u, v)$ denotes the weight of the edge $(u, v) \in E$. We model the agents' locations within the

continuous space of the edges to capture varying edge weights. Let l_t^i denote the location of agent i at time t, represented as a tuple $l_t^i = (u, v, \delta)$, indicating that the agent is situated on the edge (u, v) at a distance δ from vertex u, where $0 \le \delta \le \omega(u, v)$.

In UNSGs, the player set includes pursuers and evaders, both modeled as agents moving over the road network G. Each side may consist of a single agent or multiple agents. For example, multiple pursuers may cooperate to capture a single evader or a team of evaders. Formally, $N = (\mathbf{p}, \mathbf{e})$, where $\mathbf{p} = (p_1, p_2, \ldots, p_n), n \ge 1$ denotes pursuers and $\mathbf{e} = (e_1, e_2, \ldots, e_m), m \ge 1$ denotes evaders.

Since pursuers can block all exit nodes within a finite time, we model this duration as a predefined lockdown horizon T, which also serves as the maximum time horizon of the game. The game proceeds in discrete unit time steps. We adopt a hybrid decision-making mechanism, under which an agent makes decisions under two conditions: (1) at the beginning of each time step, all agents observe the information and make decisions simultaneously; and (2) instantaneously upon reaching a vertex during movement execution. Therefore, the state-dependent action set $A(l_t^i)$ allows an agent to move to a neighboring vertex when at a vertex, or to continue moving or reverse direction when on an edge. The game proceeds until a termination condition is met. Consequently, the number of decision points can exceed T, although the game lasts at most T discrete time steps. An evader is considered captured when its distance to any pursuer satisfies $d(l_t^{e_j}, l_t^{p_k}) \le \epsilon$, where $d(\cdot)$ represents shortest path distance between the respective points on the graph and ϵ is a predefined capture radius. The condition is evaluated at each decision point. Termination occurs when: (i) **Capture**: all evaders are captured; (ii) **Escape**: an evader reaches an exit node $v \in V_{\text{exit}}$; and (iii) **Timeout**: the game reaches the horizon T. Upon termination, rewards are assigned based on the outcome: pursuers win in cases (i) and (iii), while evaders win in case (ii).

2.2 Information and Strategy

In different real-world cases, the pursuer and evader may access different information, such as the location information of each player. With the aid of tracking devices, such as the StarChase GPS-based system [19], police officers may get the real-time location of the criminal. In another case, to avoid the worst case, the police officers usually assume that the criminal can get the real-time location of the police officers since they may not know the criminal's ability. Therefore, in our GraphChase platform, we can simulate the following four cases: i) the evader can get the real-time location information of the pursuer while the pursuer cannot get the real-time location information of the evader; ii) the pursuer can get the real-time location information of the evader while the evader cannot get the real-time location information of the pursuer; iii) both the evader and the pursuer can get the real-time location information of the opponent; and iv) both the evader and the pursuer cannot get the real-time location information of the opponent. Another important factor to consider is the communication among pursuers. Specifically, if communication is not allowed during gameplay, each pursuer must make decisions independently. In contrast, if communication is permitted, pursuers can coordinate their actions, and thus may be modeled collectively as a single player in the game.

To define the strategy for each player, we take the case where communication is permitted as an illustrative example. Based on the available real-time location information, the behavior strategy $\sigma_{\mathbf{e}}$ or $\sigma_{\mathbf{p}}$ is defined as a function that maps each decision point to a probability distribution over the corresponding set of available actions. A strategy profile σ is then defined as a tuple consisting of one strategy for each player, i.e., $\sigma = (\sigma_{\mathbf{p}}, \sigma_{\mathbf{e}})$. The pursuer's payoff function is defined by $u_{\mathbf{p}}(\sigma_{\mathbf{p}}, \sigma_{\mathbf{e}}) \in \mathbb{R}$, and the game is zero-sum, such that the evader's payoff satisfies $u_{\mathbf{e}}(\sigma_{\mathbf{p}}, \sigma_{\mathbf{e}}) = -u_{\mathbf{p}}(\sigma_{\mathbf{p}}, \sigma_{\mathbf{e}})$. We adopt the Nash equilibrium (NE) [40] as the solution concept since the NE strategy profile is a steady state in which no player can increase its utility by unilaterally deviating. In GraphChase platform, we consider the NE strategy of the pursuer to be the optimal strategy and take the worst-case utility of the pursuer as the measure for the pursuer's strategy, i.e., Worst-Case Utility$_p$ = $\max_{\sigma_{\mathbf{p}} \in \Sigma_{\mathbf{p}}} \min_{\sigma_{\mathbf{e}} \in \Sigma_{\mathbf{e}}} u_{\mathbf{p}}(\sigma_{\mathbf{p}}, \sigma_{\mathbf{e}})$.

2.3 Challenges

Computational Limits of Large Strategy Space. The network-based nature of the environment leads to a strategy space for players in UNSGs that cannot be fully enumerated due to the memory constraints [27, 65]. Specifically, when a player's strategy consists of selecting a path, the number of possible paths in large-scale UNSGs makes exhaustive enumeration infeasible. Even in relatively simple scenarios—such as when time dynamics are ignored, and pursuers can coordinate their actions—solving UNSGs remains extremely challenging [27]. Consequently, we could expect even greater difficulties in more complex settings.

Partial Observation and Imperfect Information. UNSGs typically feature imperfect information and asymmetric knowledge among participants. For example, pursuers may not have full knowledge of the evader's location or intended actions, while the evader may also possess only limited information about the pursuers. Furthermore, partial observability and uncertain time horizons further complicate the decision-making process. These difficulties highlight the need for robust algorithms capable of operating effectively under conditions of uncertainty.

Competition and Cooperation among Agents. UNSGs involve adversarial competition between pursuers and the evader, where the success of one side corresponds to the loss of the other. At the same time, pursuers cooperate within their team under a shared utility function to maximize the probability of capturing the evader. This intrinsic interplay between competition and cooperation increases overall complexity and calls for algorithmic solutions that balance optimal adversarial strategies with effective team coordination.

These challenges make UNSGs an ideal testbed for evaluating algorithms in complex and dynamic environments. The lack of a unified platform in UNSGs research motivated us to develop GraphChase as a standardized benchmark for future studies.

3 Platform: GraphChase

As shown in Figure 1, GraphChase provides template scripts to facilitate quick setup. After user configuration, the platform executes training and evaluation procedures. We next describe the core components of GraphChase, as shown in Figure 2. A brief usage guide is available in Appendix G.

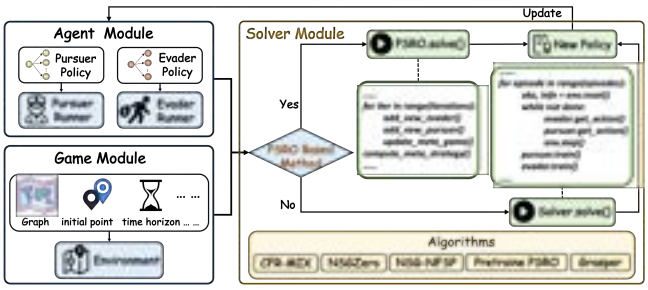

Figure 2: The core structure and workflow of GraphChase.

3.1 Core Components

GraphChase is designed to support the construction of diverse graph structures and allow custom algorithms to be implemented for solving various UNSGs. To achieve this, the platform adopts a modular architecture that decouples its core functions into three main components: Game (environment), Agent (simulated agents), and Solver (algorithm module). Under this architecture, users can define custom problem settings, use built-in baseline algorithms provided by the platform, or implement their own methods, thereby improving research efficiency.

Game Module. The Game module implements the UNSG environment (graph construction, simulations, and interfaces) and supports the following features:

(1) *Graph Structure Definition*: GraphChase represents road networks as NetworkX graphs. Users can construct graphs from an adjacency matrix or import a NetworkX `gpickle` file.

(2) *Basic Parameter Configuration*: Game parameters, such as the number of pursuers and evaders, initial positions, exit nodes, and the maximum time horizon, can be specified during initialization. Details of the configurable parameters are provided in Appendix E.

(3) *Weighted Graphs and State Representation*: GraphChase supports weighted road networks by associating each edge $(u, v) \in E$ with a weight $\omega(u, v) > 0$, representing the continuous travel time required to traverse that edge. Edge weights are either loaded from the NetworkX edge attribute or constructed from a provided adjacency matrix. To account for varying edge weights within the discrete-time game, we model agent locations in continuous space. Each agent i is represented by a tuple $x^i = (u, v, \delta)$, where $u, v \in V$ denote the endpoints of the current edge and $0 \le \delta \le \omega(u, v)$ is the distance from vertex u. A node position at vertex v is encoded as $(v, v, 0)$. The maximum time horizon T serves as a global temporal constraint in the same units as the edge weights. Each movement consumes a continuous time budget equivalent to the weight on the edges. A Timeout occurs if the cumulative travel time reaches T before capture or escape conditions are met.

(4) *Continuous-Time Step with a Unit Budget*: GraphChase uses a continuous-time transition model with a fixed time budget of 1.0 per environment step. In each call to `Env.step()`, the simulator advances by a duration $\Delta t \in (0, 1]$, where Δt is limited by the remaining time budget in the current step. All agents are updated synchronously: they move for Δt along their chosen directions, and for an agent i on an edge, its remaining distance d^i decreases by Δt. If d^i becomes 0, the agent reaches the endpoint and is encoded as $(v, v, 0)$. As a result, traversing an edge with weight $\omega(u, v) > 1$ requires multiple environment steps.

(5) *Actions, Termination, and Observations*: Actions are node IDs (start from 1) with a reserved action 0 for stay. If an agent is at a node v (i.e., $(v, v, 0)$), it may choose any neighbor $w \in \mathcal{N}(v)$ (or stay) to start traversing (v, w). If an agent is on an edge (u, v), it may choose to continue toward the destination, reverse direction toward the start node, or stay, corresponding to actions in $\{0, u, v\}$. GraphChase employs a distance-based criterion for termination. An evader is considered caught if the shortest path distance to any pursuer satisfies $d(\cdot) \le \epsilon$, where ϵ is a predefined capture radius. Escape is achieved if all evaders reach target exit nodes $v \in V_{\text{exit}}$. To accommodate continuous edge movement, we adopt a hybrid checking strategy, evaluating capture and escape conditions both at discrete time steps and whenever an agent reaches a vertex. The episode terminates upon Capture (all evaders are caught), Escape, or Timeout (the maximum time horizon T is reached). The environment provides a joint observation containing the full states of both sides (including specific edge positions); users can implement alternative information structures by wrapping the environment or customizing the observation function.

Agent Module. The Agent module provides reusable implementations of agent policies and environment-interaction runners. It is designed to decouple (i) policy representation (neural or non-neural), (ii) trajectory collection and training pipelines, and (iii) optimization logic (delegated to the Solver module). It revolves around two core abstractions: `Policy` and `Runner`.

(1) *Unified Policy Interface:* The module defines a lightweight interface for action selection and value estimation. This unified design allows solvers to treat heterogeneous policies uniformly—whether they are deep neural networks (e.g., PPO) or structured heuristics (e.g., path-search algorithms). This flexibility is critical for UNSGs, where baselines often mix learning-based and rule-based strategies.

(2) *Runner-Based Abstraction:* To decouple the solver from low-level simulation loops, we introduce `Runners` to encapsulate environment interaction, preprocessing, and batch construction. Runners handle observation encoding and action masking, exposing only processed tensors to the agents. This design also supports vectorized rollouts, significantly accelerating data collection.

(3) *Support for Iterative Game-Theoretic Learning:* The module is explicitly designed for iterative solvers like PSRO. It supports efficient policy cloning and persistence, enabling the creation of policy populations and best response training without re-implementing environment logic or model architectures.

Solver Module. The Solver module implements a suite of learning based and game-theoretic solvers for UNSGs, and is responsible for (i) updating agent parameters through reusable optimization algorithms and (ii) orchestrating higher-level equilibrium-finding procedures (e.g., PSRO) that iterate between best response computation and meta-strategy updates. It is decoupled from the game mechanics, interacting with agents solely via the standardized Runner interfaces. The Solver module supports the following features:

(1) *Plug-and-Play Optimizers*: GraphChase provides reusable optimizers such as PPO and MAPPO for best response training. These algorithms consume batches prepared by runners and update agent parameters without interacting with the environment. Consequently, a new environment or a new policy network can reuse the

optimizer as long as the runner outputs the expected batch fields (e.g., action indices, log-probabilities, rewards, and value targets).

(2) *Support for Game-Theoretic Learning (PSRO)*: GraphChase supports Policy Space Response Oracles (PSRO). The framework decomposes PSRO into modular components: (i) a **Best Response Oracle** that can leverage either RL (e.g., PPO and MAPPO) or heuristic search, and (ii) a **Meta-Solver** (e.g., Projected Replicator Dynamics) that operates on payoff matrices. This modularity enables researchers to apply the PSRO framework via the GraphChase interface, eliminating redundant code. It also allows researchers to experiment with custom meta-solvers and best response oracles.

(3) *Support for Custom Solvers*: Beyond baselines like PSRO and other benchmark algorithms, GraphChase supports custom algorithm implementation. Users can define their own solvers via our unified interface without modifying the underlying game core.

3.2 Benchmark Algorithms and Evaluation

To establish a benchmark for researchers, GraphChase integrates several representative algorithms for UNSGs. (1) **CFR-MIX** algorithm [34] enhances traditional counterfactual regret minimization (CFR) [66] with deep learning techniques [9]. (2) **NSG-NFSP** [53] adapts the neural fictitious self-play method [20] by producing action-value outputs for the best response policy and transforming the average policy into a classifier that assigns distributions only over valid actions. (3) **NSGZero** [52] develops deep neural networks to conduct neural Monte Carlo tree search, supporting efficient joint network training. Variants of the PSRO framework include: (4) **Pretrained PSRO** [33] initially trains the pursuer's model against a variety of evader strategies before refining it with the PSRO loop to improve best response performance; and (5) **Grasper** [32] leverages a graph neural network to represent the graph structure as hidden vectors and utilizes a hypernetwork to generate pursuer policies from these representations. It then follows a three-phase process: pre-pretraining for robust game encoding, multi-task pre-training for stabilizing policy learning, and PSRO-based fine-tuning.

Evaluation. In current UNSGs research domain, evaders make decisions based on valid paths or source-target pairs. Specifically, before the pursuit-evasion process begins, the evader selects one target from a set of feasible exits. The evader then chooses a path from its initial position to the selected target and follows this predetermined path at each time step. The benchmark algorithms all adopt this decision-making approach for the evader. As such, all current experiments are conducted under this setting. Therefore, the primary objective is to evaluate the optimal strategy for the pursuer. GraphChase provides three evaluation methods.

- **Worst-Case Utility.** This method enumerates all possible paths the evader might take and then evaluates the utility of the pursuer along each one. The pursuer's utility on the path that yields the lowest possible value is taken as the worst case. This provides a measure of the pursuer's performance under the most disadvantageous evasion strategy.
- **Pseudo Worst-Case Utility.** As the size of the graph increases, exhaustive enumeration of all evader paths becomes time consuming. To address this, we use pseudo worst-case utility as an approximation to the worst-case exit setting. For each target exit, several paths leading to this exit are randomly sampled, and the average pursuer utility over these paths is used to estimate the pursuer's expected utility conditioned on that exit. We take the minimum value across all exits as the pseudo worst-case utility.
- **Visualization.** GraphChase supports process visualization. Visualizing trajectories and decision-making processes allows researchers to assess whether a strategy aligns with human intuition. An example is provided in Appendix C.

4 Experimental Evaluation

In this section, we evaluates the correctness, efficiency, and versatility of GraphChase. We use the platform to benchmark SOTA algorithms on various real-world maps and analyze their limitations regarding robustness and scalability. Throughout the experiments in this section, we report pseudo worst-case utility and denote it as WCU for brevity. All experiments were performed on a server with a 48-core Intel Xeon Gold 6248R CPU and 8 NVIDIA A30 GPUs[1].

4.1 Reproducibility and Correctness Validation

A primary objective of GraphChase is to ensure that implemented algorithms reproduce results from their original literature[2]. To verify this, we evaluate Pretrained PSRO, Grasper, NSGZero, NSG-NFSP, and CFR-MIX on undirected grid graphs of 5×5 and 7×7. Each setting involves 4 pursuers and 1 evader. By designing exit locations and agent starting positions, we established two difficulty levels per graph size, corresponding to ground-truth capture probabilities (equivalently, the pursuer's worst-case utility) of 0.5 and 1.0. See Appendix A for how these ground-truth values are computed. These settings are denoted by their grid size and probability, where $7(0.5)$ represents a 7×7 grid with a capture probability of 0.5.

Figure 3 illustrates the training dynamics of the pursuer's worst-case utility for Pretrained PSRO, Grasper, and NSG-NFSP against wall-clock training time on the 7×7 grid. Using the same hyperparameters as the original studies, our implementations in GraphChase exhibit convergence trajectories that align closely with the original baselines. GraphChase resolves implementation inefficiencies in the original Grasper codebase, particularly the partial use of collected batch data during updates, resulting in better training performance while maintaining consistent logic.

Table 1 summarizes the converged performance for all five algorithms across the four game settings. The final worst-case utility values are generally consistent with those reported in the original codebases, except for Grasper. This consistency across different graph scales and difficulty levels validates that GraphChase replicates the performance of the reference implementations. The detailed structures and visualizations of the maps used in our experiments can be found in Appendix B.

4.2 Computational Efficiency Analysis

Figure 3 shows that under the same hyperparameters, GraphChase reduces time overhead for training process. Using Pretrained PSRO as a case study, we compared the efficiency of GraphChase against original implementations on the $7(0.5)$ setting, with identical hyperparameters and the same GPU/CPU setup, across three metrics:

[1]Our code is available at https://github.com/GraphChase/GraphChasePlatform.git
[2]Codes were shared by the original authors of these algorithms.

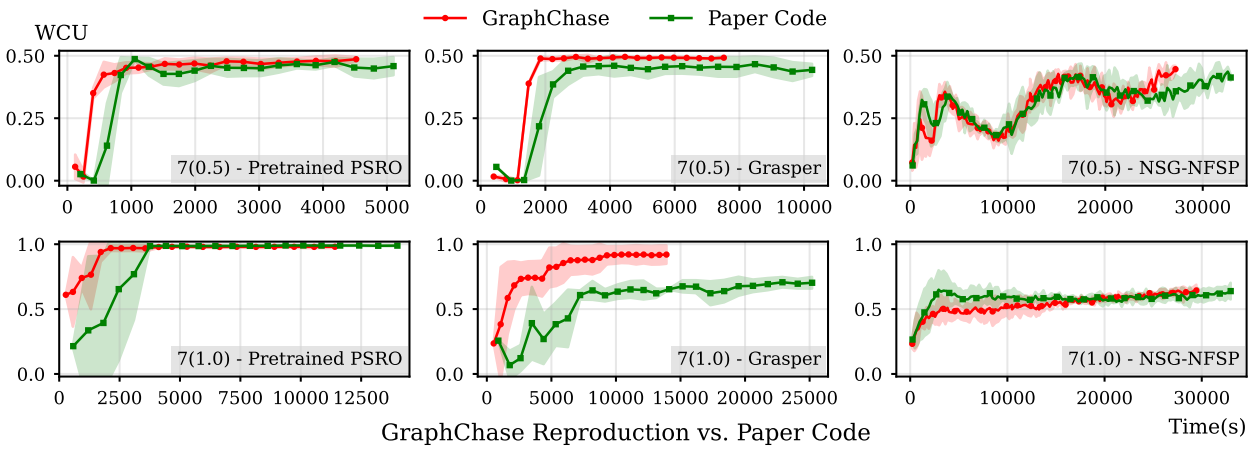

Figure 3: Pursuer pseudo worst-case utility curves during training on the 7×7 grid graph. Each figure title specifies the game configuration and the algorithm. The x-axis denotes the training duration in seconds. Solid lines represent the mean worst-case utility, while shaded regions indicate the standard deviation calculated across 3 independent seeds.

Algorithm	Method	7(0.5)	7(1.0)	5(0.5)	5(1.0)
Pretrained PSRO	GraphChase	0.4855 ± 0.0055	0.9803 ± 0.0008	0.4928 ± 0.0088	0.9752 ± 0.0012
	Paper Code	0.4535 ± 0.0040	0.9891 ± 0.0007	0.4740 ± 0.0026	0.9891 ± 0.0001
Grasper	GraphChase	0.4845 ± 0.0028	0.9779 ± 0.0017	0.4547 ± 0.0059	0.9515 ± 0.0025
	Paper Code	0.4446 ± 0.0068	0.7857 ± 0.0028	0.4147 ± 0.0115	0.9691 ± 0.0033
NSGZero	GraphChase	0.3268 ± 0.0571	0.9997 ± 0.0003	0.4673 ± 0.0233	0.9999 ± 0.0001
	Paper Code	0.3375 ± 0.0293	0.9927 ± 0.0004	0.4634 ± 0.0295	0.9876 ± 0.0003
NSG–NFSP	GraphChase	0.4357 ± 0.0086	0.6374 ± 0.0060	0.4415 ± 0.0364	0.8457 ± 0.0154
	Paper Code	0.4243 ± 0.0097	0.6286 ± 0.0077	0.4447 ± 0.0237	0.7803 ± 0.0237
CFR–MIX	GraphChase	0.212 ± 0.0281	0.225 ± 0.0106	0.4167 ± 0.0219	0.5497 ± 0.0168
	Paper Code	0.217 ± 0.0384	0.229 ± 0.0201	0.4115 ± 0.0287	0.5354 ± 0.0206

Table 1: Comparison of converged pursuer pseudo worst-case utility across four game settings. Experiments were conducted using identical hyperparameters to the original baselines. For each run, the utility is averaged over the stable convergence phase; the reported values present the aggregate mean ± standard deviation across three different random seeds.

the time to compute the Evader Best Response (BR), Pursuer BR, and sampling throughput measured in Episodes Per Second (EPS).

As shown in Figure 4, GraphChase improves the overall sampling throughput by 1.38×, indicating more episodes can be generated per second under the same hardware. Moreover, GraphChase accelerates best response computation on both sides: the evader BR is sped up by 1.96× and the pursuer BR by 1.72×. This efficiency stems from the vectorized environment design and optimized graph operations, making the platform suitable for large-scale training. More details can be found in Appendix F.

4.3 Real-World Urban Topology Benchmarks

To evaluate the generalization capabilities of current UNSG algorithms, we established a leaderboard using real-world urban topologies provided by GraphChase. We selected six distinct maps of varying scales and complexities: Singapore, Manhattan, Mumbai, Scotland Yard, Times Square, and the Sydney Opera House. These topologies were sourced from *OpenStreetMap* and abstracted into graph structures. We modeled these topologies as unweighted, undirected graphs to align with the existing state-of-the-art algorithms and ensure a fair evaluation baseline.

Table 2 presents the benchmark results across real-world maps. The results reveal performance variance depending on the topology. While most algorithms achieve high pursuer worst-case utility, the utility values are lower in environments such as Manhattan and Times Square, which are characterized by higher numbers of nodes, edges, and extended time horizons. These results provide a baseline for evaluating UNSG solutions on realistic networks.

4.4 Robustness Evaluation

In addition to standard performance metrics, we evaluate the robustness of learned policies against environmental shifts.

Robustness to Horizon Shift. We investigate the robustness of policies to changes in the max time horizon (T). Pursuer policies

	Grasper	Pretrained PSRO	NSGZero	NSG-NFSP	Nodes	Edges	T
Manhattan	0.5897 ± 0.0540	0.1311 ± 0.0033	0.0057 ± 0.0003	0.4108 ± 0.0933	620	1081	13
Mumbai	0.9923 ± 0.0009	0.9267 ± 0.0188	0.9981 ± 0.0011	0.7008 ± 0.0302	71	92	8
Singapore	0.9972 ± 0.0003	0.9254 ± 0.0100	0.9982 ± 0.0001	0.8004 ± 0.0192	158	212	12
Scotland Yard	0.9876 ± 0.0013	0.1573 ± 0.0032	0.9981 ± 0.0007	0.5531 ± 0.0390	200	391	9
Sydney Opera House	0.9956 ± 0.0004	0.9640 ± 0.0023	0.9879 ± 0.0289	0.8536 ± 0.0243	65	87	7
Times Square	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0049 ± 0.0003	0.0000 ± 0.0000	242	421	16

Table 2: Performance comparison of benchmark algorithms on real-world maps. The reported values represent the pursuer's worst-case utility at convergence, presented as mean ± standard deviation. The columns *Nodes* and *Edges* indicate the number of vertices and undirected connections in each graph, respectively. T denotes the maximum time horizon for each game.

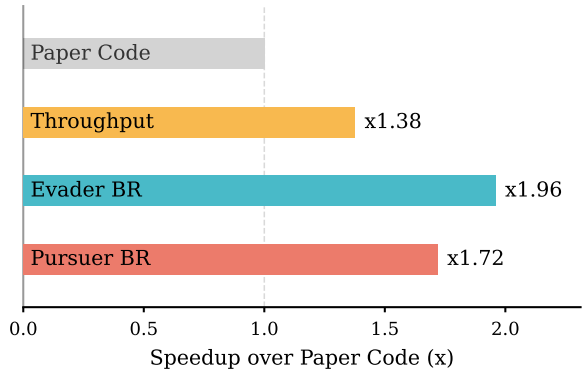

Figure 4: Speedup of GraphChase over the paper code across throughput and best response computation. The throughput ratio is calculated by dividing the EPS of GraphChase by the EPS of the paper code. Similarly, the speedup for best response computation is determined by the ratio of the execution time of the paper code to that of GraphChase. The dashed vertical line indicates the paper code baseline (×1.0); values larger than 1 denote faster execution.

were trained on 7(0.5) and 7(1.0) settings with $T = 4$, then frozen and tested at an extended horizon of $2T$. As shown in Table 3, the pursuer's worst-case utility decreases under horizon extension, indicating limited robustness to horizon shifts.

Robustness to Edge Weight Variations. Motivated by urban dynamics, we examine robustness against structural variations. While the topology of a network remains static, traffic conditions change, altering the travel time of edges. To simulate cost variations, we apply static reweighting by randomly assigning travel-time costs to edges and using symmetric weights ($w(u, v) = w(v, u)$), fixed within each episode. We test policies trained on unweighted graphs under the weighted setting. As shown in Table 4, the pursuer's utility drops significantly in the weighted setting, thereby underscoring limited robustness to edge-cost heterogeneity.

4.5 Scalability Evaluation

To evaluate the scalability of existing algorithms, we apply benchmark methods to solve larger game instances. Specifically, we conduct experiments on a 100×100 grid graph with a maximum time horizon of $T = 200$. In this setting, four pursuers aim to capture a single evader, who seeks to evade capture by escaping through one of twelve exit nodes. We observed that none of the algorithms were able to complete the training process despite running the experiments for several days. In practice, even on a 30×30 grid, after two days of training, the run still stalled at the path-enumeration stage.

This failure stems from the reliance on exhaustive enumeration of the evader action space in current methods. These methods require pre-calculating all simple paths from the evader's starting position to the exits within the maximum time horizon (T) to facilitate decision-making during simulations. As the graph size and time horizon increase, the number of valid paths grows exponentially leading to computational intractability. Figure 9 illustrates the time cost required to enumerate simple paths from the center of a grid to a corner across varying grid sizes. The computational overhead becomes large once the grid size exceeds 20×20. This exponential explosion in path enumeration time constitutes the scalability barrier for current UNSG solvers and suggests that new approaches must move away from global action space traversal. More details can be found in Appendix D.

5 Related Works

Game theory has emerged as a valuable tool in addressing complex interactions and has been successfully applied to various security challenges [27, 37, 47], including allocating limited resources to protect infrastructure [26] or designing patrolling strategies in adversarial settings [51]. Behind these results, one important model is Stackelberg Security Games (SSGs), which is used to solve a variety of security problems [47]. In SSGs, the defender moves first and then the attacker best responds to the defender's strategy. Then, the UNSGs model is a special case of SSG, which is used in the zero-sum environment on networks.

UNSGs share similarities with pursuit-evasion games [43], which involve strategic interactions between multiple pursuers and one or more evaders within a defined environment [7]. Pursuit-evasion games are classic research problems with applications in areas ranging from civilian safety [41] to military operations [49], and have been investigated in fields such as physics [25], mathematics [28, 42], and engineering [15]. Pursuit-evasion games were formulated as differential games, with foundational work by Rufus Isaacs in "Differential Games" [25]. Subsequent studies developed various algorithms, such as the linear–quadratic differential game

		Pretrained PSRO	Grasper	NSGZero	NSG-NFSP
7(0.5)	T=4	0.477	0.481	0.396	0.466
	T=8	0.385	0.236	0.220	0.107
7(1.0)	T=4	0.996	0.989	0.741	0.982
	T=8	0.077	0.195	0.145	0.267

Table 3: Robustness to horizon shift. The table shows the pursuer pseudo worst-case utility when the max time horizon is extended from $T = 4$ to $T = 8$.

		Pretrained PSRO	Grasper	NSG-NFSP
7(0.5)	undirected graph	0.483	0.484	0.435
	weighted graph	0.293	0.343	0.159
7(1.0)	undirected graph	0.983	0.978	0.679
	weighted graph	0.111	0.259	0.254

Table 4: Robustness to edge weight variations. Comparison of pursuer pseudo worst-case utility between unweighted undirected, and statically weighted version with the same connectivity. Edge weights are randomly sampled and fixed within each episode. NSGZero is excluded here as its online MCTS assumes unweighted rollouts and is not directly applicable to our weighted transfer setting.

(LQDG) approach by Ho et al. [21], and the use of graphs to describe these problems by Parsons [43]. Over time, the field expanded to include discrete settings, with work addressing discrete-time multiple-pursuer single-evader games [8], and one-sided partially observable scenarios where the evader is fully informed, but the pursuers are not [22–24]. Related domains such as patrolling security games (PSGs) have also been explored, where defenders guard against unseen intruders in stochastic, often infinite-horizon settings [6, 51]. PSGs have been extended to accommodate uncertain signals and coordinated response [3, 5]. Our GraphChase can be extended to cover these settings.

Existing multiplayer benchmarks for pursuit-evasion games, such as SIMPE [48], MARBLER [27], and Avalon [1], have enriched the field with various testbeds. SIMPE explores diverse strategies with multiple pursuers and one evader, but operates in a continuous space and does not account for temporal constraints, which are essential in UNSGs. MARBLER links MARL with physical robots, while Avalon focuses on procedural worlds to evaluate generalization, primarily simulating biological survival rather than explicit pursuit-evasion. Similarly, environments like Google Research Football [29] and Starcraft [45] provide MARL benchmarks in plane settings. However, these platforms emphasize algorithmic development and often overlook game-theoretic dynamics relevant to pursuit-evasion. Openspiel [30] offers a wide range of games but does not include pursuit-evasion scenarios. As a result, a unified benchmark for modeling UNSGs remains lacking. Our GraphChase fills this gap by providing a flexible environment for UNSGs.

6 Discussion: Testbed for Multiplayer Games

Computing an NE in multiplayer games is generally hard [14, 62], and designing efficient algorithms for computing such an NE is still an open challenge. Our platform could be a testbed for algorithms for solving multiplayer games. In particular, our platform provides real-world scenarios for adversarial team games [2, 4, 12, 13, 17, 18, 35, 36, 50, 54–61], where a group of players competes against an adversary or another team. Various solution concepts apply depending on the situation. When team players compete independently against the adversary, the relevant solution concepts include 1) NE [39, 62], where no player gains by deviating from this equilibrium, and 2) team-maxmin equilibrium (TME) [4, 13, 50, 58, 59, 61], which is a type of NE that optimizes the team's utility across all NEs. Based on our platform, if we set that pursuers independently try to interdict the evader, we can also use our platform to compute an NE or TME in normal-form or extensive-form games. For normal-form games where team players can coordinate their strategies, the applicable solution concept is the correlated team-maxmin equilibrium (CTME) [4]. This is essentially equivalent to an NE in zero-sum two-player games, as the team with coordinated strategies and a unified payoff function behaves like a single player. In extensive-form games, the team with coordinated strategies has two solution concepts: 1) team-maxmin equilibrium with a communication device (TMECom) [13], applicable when the team can continuously communicate and coordinate strategies, making the game akin to a two-player zero-sum game with perfect recall; and 2) team-maxmin equilibrium with a coordination device (TMECor) [13, 60, 64], used when the team can only coordinate strategies before gameplay, rendering the game similar to a two-player zero-sum game with imperfect recall. The algorithms in [32–34, 52, 53, 65] implemented on GraphChase compute a TMECom that is NE in team adversarial games. If we set that the team can only coordinate strategies before gameplay in the extensive-form games, we can also compute a TMECor on GraphChase.

7 Conclusion

In this paper, we propose GraphChase, the first open-source platform dedicated to UNSGs, offering researchers a flexible multiplayer game environment and aiding in developing scalable algorithms. Specifically, we develop a unified and flexible platform to accommodate different UNSGs environments and implement several deep learning-based algorithms as benchmarks. Extensive experiments on GraphChase provide insights into these algorithms. We hope that GraphChase will facilitate the establishment of standardized evaluation criteria for UNSGs algorithms, thereby contributing to both theoretical advances and practical applications in solving general multi-agent games.

Realistic Modeling: Our platform stands out for its realistic modeling of complex, real-world environments, thanks to the advanced capabilities of GraphChase. By combining powerful graph-based representations with a hybrid decision-making mechanism, it effectively captures the dynamic and unpredictable nature of real scenarios — including fluctuating traffic patterns, spontaneous human behaviors, and other variable elements that traditional models often oversimplify. Furthermore, GraphChase provides native support for weighted graphs and user-updatable costs, enabling seamless integration of data-driven, time-dependent disruptions such as real-time traffic feeds, accident-induced road closures, or sudden congestion changes. This flexibility ensures that simulations remain highly faithful to actual conditions, delivering practical and reliable insights for real-world applications.

References

[1] Joshua Albrecht, Abraham Fetterman, Bryden Fogelman, Ellie Kitanidis, Bartosz Wróblewski, Nicole Seo, Michael Rosenthal, Maksis Knutins, Zack Polizzi, James Simon, et al. 2022. Avalon: A benchmark for rl generalization using procedurally generated worlds. Advances in Neural Information Processing Systems 35 (2022), 12813–12825.

[2] Ioannis Anagnostides, Fivos Kalogiannis, Ioannis Panageas, Emmanouil-Vasileios Vlatakis-Gkaragkounis, and Stephen Mcaleer. 2023. Algorithms and Complexity for Computing Nash Equilibria in Adversarial Team Games. In EC.

[3] Nicola Basilico, Andrea Celli, Giuseppe De Nittis, and Nicola Gatti. 2017. Coordinating multiple defensive resources in patrolling games with alarm systems. In Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems. 678–686.

[4] Nicola Basilico, Andrea Celli, Giuseppe De Nittis, and Nicola Gatti. 2017. Team-maxmin equilibrium: Efficiency bounds and algorithms. In AAAI. 356–362.

[5] Nicola Basilico, Giuseppe De Nittis, and Nicola Gatti. 2017. Adversarial patrolling with spatially uncertain alarm signals. Artificial Intelligence 246 (2017), 220–257.

[6] Nicola Basilico, Nicola Gatti, Francesco Amigoni, et al. 2009. Leader-follower strategies for robotic patrolling in environments with arbitrary topologies. In Proceedings of the International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS). 57–64.

[7] Ahmet Tunc Bilgin and Esra Kadioglu-Urtis. 2015. An approach to multi-agent pursuit evasion games using reinforcement learning. In 2015 International Conference on Advanced Robotics (ICAR). IEEE, 164–169.

[8] Shaunak D Bopardikar, Francesco Bullo, and Joao P Hespanha. 2008. On discrete-time pursuit-evasion games with sensing limitations. IEEE Transactions on Robotics 24, 6 (2008), 1429–1439.

[9] Noam Brown, Adam Lerer, Sam Gross, and Tuomas Sandholm. 2019. Deep counterfactual regret minimization. In International conference on machine learning. PMLR, 793–802.

[10] Noam Brown and Tuomas Sandholm. 2018. Superhuman AI for heads-up no-limit poker: Libratus beats top professionals. Science 359, 6374 (2018), 418–424.

[11] Noam Brown and Tuomas Sandholm. 2019. Superhuman AI for multiplayer poker. Science 365, 6456 (2019), 885–890.

[12] Luca Carminati, Federico Cacciamani, Marco Ciccone, and Nicola Gatti. 2022. A marriage between adversarial team games and 2-player games: Enabling abstractions, no-regret learning, and subgame solving. In ICML. PMLR, 2638–2657.

[13] Andrea Celli and Nicola Gatti. 2018. Computational results for extensive-form adversarial team games. In AAAI. 965–972.

[14] Xi Chen and Xiaotie Deng. 2005. 3-Nash is PPAD-complete. In Electronic Colloquium on Computational Complexity, Vol. 134. 2–29.

[15] J Mikael Eklund, Jonathan Sprinkle, and S Shankar Sastry. 2011. Switched and symmetric pursuit/evasion games using online model predictive control with application to autonomous aircraft. IEEE Transactions on Control Systems Technology 20, 3 (2011), 604–620.

[16] Meta Fundamental AI Research Diplomacy Team FAIR, Anton Bakhtin, Noam Brown, Emily Dinan, Gabriele Farina, Colin Flaherty, Daniel Fried, Andrew Goff, Jonathan Gray, Hengyuan Hu, et al. 2022. Human-level play in the game of Diplomacy by combining language models with strategic reasoning. Science 378, 6624 (2022), 1067–1074.

[17] Gabriele Farina, Andrea Celli, Nicola Gatti, and Tuomas Sandholm. 2018. Ex ante coordination and collusion in zero-sum multi-player extensive-form games. In NeurIPS. 9638–9648.

[18] Gabriele Farina, Andrea Celli, Nicola Gatti, and Tuomas Sandholm. 2021. Connecting optimal ex-ante collusion in teams to extensive-form correlation: Faster algorithms and positive complexity results. In ICML. 3164–3173.

[19] Morgan Gaither, Mark Gabriele, Nancy Andersen, Sean Healy, and Vivian Hung. 2017. Pursuit technology impact assessment, Version 1.1. Final Report to the US Department of Justice (2017).

[20] Johannes Heinrich and David Silver. 2016. Deep reinforcement learning from self-play in imperfect-information games. arXiv preprint arXiv:1603.01121 (2016).

[21] Y Ho, Arthur Bryson, and Sheldon Baron. 1965. Differential games and optimal pursuit-evasion strategies. IEEE Trans. Automat. Control 10, 4 (1965), 385–389.

[22] Karel Horák and Branislav Bošanskỳ. 2016. A Point-Based Approximate Algorithm for One-Sided Partially Observable Pursuit-Evasion Games. In 7th International Conference on Decision and Game Theory for Security. 435–454.

[23] Karel Horák and Branislav Bošanskỳ. 2017. Dynamic Programming for One-sided Partially Observable Pursuit-evasion Games. In International Conference on Agents and Artificial Intelligence, Vol. 2. SCITEPRESS, 503–510.

[24] Karel Horák, Branislav Bošanskỳ, and Michal Pěchouček. 2017. Heuristic search value iteration for one-sided partially observable stochastic games. In Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 31. 558–564.

[25] R. Isaacs. 1965. Differential Games: A Mathematical Theory with Applications to Warfare and Pursuit, Control and Optimization. Wiley. https://books.google.com.sg/books?id=gtlQAAAAMAAJ

[26] Manish Jain, Vincent Conitzer, and Milind Tambe. 2013. Security Scheduling for Real-world Networks. In AAMAS (St. Paul, MN, USA). 215–222.

[27] Manish Jain, Dmytro Korzhyk, Ondřej Vaněk, Vincent Conitzer, Michal Pěchouček, and Milind Tambe. 2011. A double oracle algorithm for zero-sum security games on graphs. In AAMAS. 327–334.

[28] Swastik Kopparty and Chinya V Ravishankar. 2005. A framework for pursuit evasion games in Rn. Inform. Process. Lett. 96, 3 (2005), 114–122.

[29] Karol Kurach, Anton Raichuk, Piotr Stańczyk, Michał Zajac, Olivier Bachem, Lasse Espeholt, Carlos Riquelme, Damien Vincent, Marcin Michalski, Olivier Bousquet, et al. 2020. Google research football: A novel reinforcement learning environment. In Proceedings of the AAAI conference on artificial intelligence, Vol. 34. 4501–4510.

[30] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, et al. 2019. OpenSpiel: A framework for reinforcement learning in games. arXiv preprint arXiv:1908.09453 (2019).

[31] Marc Lanctot, Vinicius Zambaldi, Audrunas Gruslys, Angeliki Lazaridou, Karl Tuyls, Julien Pérolat, David Silver, and Thore Graepel. 2017. A unified game-theoretic approach to multiagent reinforcement learning. In NeurIPS. 4190–4203.

[32] Pengdeng Li, Shuxin Li, Xinrun Wang, Jakub Cerny, Youzhi Zhang, Stephen McAleer, Hau Chan, and Bo An. 2024. Grasper: A Generalist Pursuer for Pursuit-Evasion Problems. AAMAS (2024).

[33] Shuxin Li, Xinrun Wang, Youzhi Zhang, Wanqi Xue, Jakub Černỳ, and Bo An. 2023. Solving large-scale pursuit-evasion games using pre-trained strategies. In AAAI, Vol. 37. 11586–11594.

[34] Shuxin Li, Youzhi Zhang, Xinrun Wang, Wanqi Xue, and Bo An. 2021. CFR-MIX: Solving imperfect information extensive-form games with combinatorial action space. In IJCAI. 3663–3669.

[35] Shuxin Li, Youzhi Zhang, Xinrun Wang, Wanqi Xue, and Bo An. 2023. Decision Making in Team-Adversary Games with Combinatorial Action Space. CAAI Artificial Intelligence Research 2 (2023).

[36] Stephen Marcus McAleer, Gabriele Farina, Gaoyue Zhou, Mingzhi Wang, Yaodong Yang, and Tuomas Sandholm. 2023. Team-PSRO for Learning Approximate TMECor in Large Team Games via Cooperative Reinforcement Learning. In NeurIPS.

[37] Sara Marie McCarthy, Milind Tambe, Christopher Kiekintveld, Meredith L Gore, and Alex Killion. 2016. Preventing illegal logging: Simultaneous optimization of resource teams and tactics for security.. In AAAI. 3880–3886.

[38] Matěj Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. 2017. DeepStack: Expert-level artificial intelligence in no-limit poker. Science 356 (2017), 508–513.

[39] John Nash. 1951. Non-cooperative games. Annals of Mathematics (1951), 286–295.

[40] John F Nash. 1950. Equilibrium points in n-person games. PNAS 36, 1 (1950), 48–49.

[41] Dave W Oyler, Pierre T Kabamba, and Anouck R Girard. 2016. Pursuit–evasion games in the presence of obstacles. Automatica 65 (2016), 1–11.

[42] M Pachter. 1987. Simple-motion pursuit-evasion in the half plane. Computers & Mathematics with Applications 13, 1-3 (1987), 69–82.

[43] TD Parsons. 1976. Pursuit-Evasion in a Graph. Theory and Applications of Graphs (1976).

[44] Frederick P Rivara and Christopher D Mack. 2004. Motor vehicle crash deaths related to police pursuits in the United States. Injury Prevention 10, 2 (2004), 93–95.

[45] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder De Witt, Gregory Farquhar, Nantas Nardelli, Tim GJ Rudner, Chia-Man Hung, Philip HS Torr, Jakob Foerster, and Shimon Whiteson. 2019. The starcraft multi-agent challenge. arXiv preprint arXiv:1902.04043 (2019).

[46] Yoav Shoham and Kevin Leyton-Brown. 2008. Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations. Cambridge University Press.

[47] Arunesh Sinha, Fei Fang, Bo An, Christopher Kiekintveld, and Milind Tambe. 2018. Stackelberg Security Games: Looking Beyond a Decade of Success.. In IJCAI. 5494–5501.

[48] Shahriar Talebi and Marwan A Simaan. 2018. SIMPE: A Simulation Platform for Multi-Player Pursuit-Evasion Problems. In 2018 IEEE 14th International Conference on Control and Automation (ICCA). IEEE, 344–349.

[49] Bogdan Vlahov, Eric Squires, Laura Strickland, and Charles Pippin. 2018. On developing a uav pursuit-evasion policy using reinforcement learning. In 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA). IEEE, 859–864.

[50] Bernhard von Stengel and Daphne Koller. 1997. Team-maxmin equilibria. Games and Economic Behavior 21, 1-2 (1997), 309–321.

[51] Yevgeniy Vorobeychik, Bo An, Milind Tambe, and Satinder Singh. 2014. Computing solutions in infinite-horizon discounted adversarial patrolling games. In Proceedings of the International Conference on Automated Planning and Scheduling, Vol. 24. 314–322.

[52] Wanqi Xue, Bo An, and Chai Kiat Yeo. 2022. NSGZero: Efficiently learn non-exploitable policy in large-scale network security games with neural mc carlo tree search. In AAAI. 4646–4653.
[53] Wanqi Xue, Youzhi Zhang, Shuxin Li, Xinrun Wang, Bo An, and Chai Kiat 2021. Solving large-scale extensive-form network security games via ne fictitious self-play. In IJCAI. 3713–3720.
[54] Brian Zhang, Luca Carminati, Federico Cacciamani, Gabriele Farina, Pie cardo Olivieri, Nicola Gatti, and Tuomas Sandholm. 2022. Subgame solvin adversarial team games. In NeurIPS. 26686–26697.
[55] Brian Hu Zhang, Gabriele Farina, Andrea Celli, and Tuomas Sandholm. 2022. C mal correlated equilibria in general-sum extensive-form games: Fixed-param algorithms, hardness, and two-sided column-generation. In EC. 1119–1120.
[56] Brian Hu Zhang, Gabriele Farina, and Tuomas Sandholm. 2023. Team Be DAG: Generalizing the Sequence Form to Team Games for Fast Computatio Correlated Team Max-Min Equilibria via Regret Minimization. In ICML. 409 41018.
[57] Brian Hu Zhang and Tuomas Sandholm. 2022. Team correlated equilibri zero-sum extensive-form games via tree decompositions. In AAAI. 5252–52
[58] Youzhi Zhang and Bo An. 2020. Computing team-maxmin equilibria in zero- multiplayer extensive-form games. In AAAI. 2318–2325.
[59] Youzhi Zhang and Bo An. 2020. Converging to team-maxmin equilib zero-sum multiplayer games. In ICML. 11033–11043.
[60] Youzhi Zhang, Bo An, and Jakub Černỳ. 2021. Computing ex ante coordi team-maxmin equilibria in zero-sum multiplayer extensive-form games. In Vol. 35. 5813–5821.
[61] Youzhi Zhang, Bo An, and V.S. Subrahmanian. 2022. Correlation-based algo for team-maxmin equilibrium in multiplayer extensive-form games. In 606–612.
[62] Youzhi Zhang, Bo An, and Venkatramanan Subrahmanian. 2023. Comp Optimal Nash Equilibria in Multiplayer Games. NeurIPS 36.
[63] Youzhi Zhang, Bo An, Long Tran-Thanh, Zhen Wang, Jiarui Gan, and Nich Jennings. 2017. Optimal escape interdiction on transportation networks. In 3936–3944.
[64] Youzhi Zhang, Bo An, and Daniel Dajun Zeng. 2024. DAG-based column g tion for adversarial team games. ICML.
[65] Youzhi Zhang, Qingyu Guo, Bo An, Long Tran-Thanh, and Nicholas R Jen 2019. Optimal interdiction of urban criminals with the aid of real-time inf tion. In AAAI, Vol. 33. 1262–1269.
[66] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Pic 2008. Regret minimization in games with incomplete information. In Ne 1729–1736.

A Description of Experimental Settings

The graph structures for the 5×5 and 7×7 grids are illustrated in Figure 5 and 6.

The reason why the left side of Figure 6 reaches a probability of 0.5 under the Nash equilibrium is explained as follows:

Exit Node Accessibility: Among the available exit nodes, two (located at the bottom-left and bottom-right) require more than three steps for the evader to reach. However, the pursuers can reach these exits in exactly three steps. As a result, a rational evader will avoid these exits, since choosing them would guarantee capture.

Viable Exit Options: This constraint leaves only two viable exit nodes for the evader. Only the pursuer positioned at the top-left corner is able to reach either of these exits within three steps. Furthermore, both exits present equal strategic value for both the evader and the pursuer.

Probability Calculation: Rational decision-making leads the evader to choose between these two viable exits with equal probability (i.e., 1/2 for each). For each exit, the probability of being caught depends on whether the pursuer also chooses the same exit, which is again 1/2. Thus, the total probability is determined as: 2 exits $\times$(1/2 chance of the evader choosing each exit) $\times$ (1/2 chance of the pursuer selecting the same exit) = 0.5.

A similar reasoning process applies to the scenario on the left side of Figure 5. Under the Nash equilibrium, the probability that the evader is captured is also 0.5.

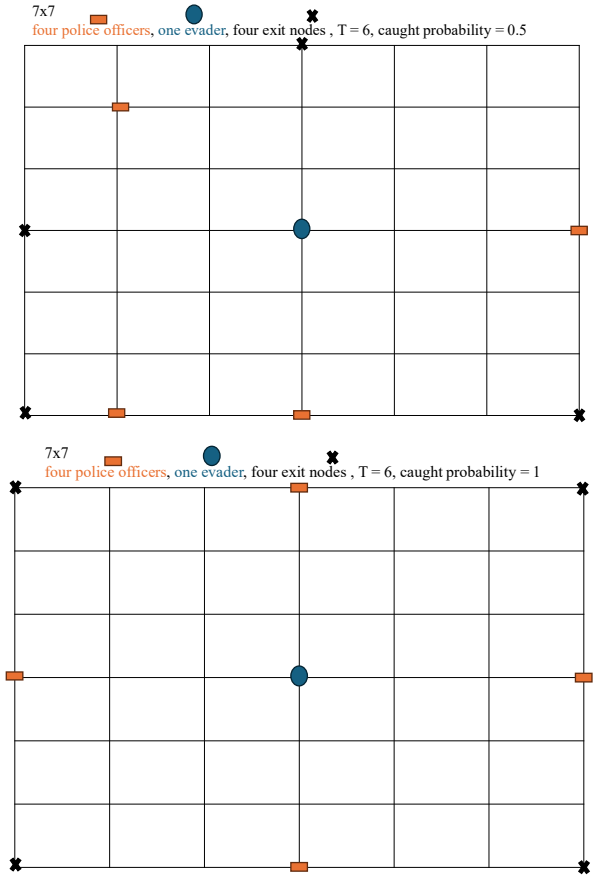

Figure 5: 7×7 grid graph settings with the evader caught probability of 0.5 (left) or 1 (right).

B Detailed Visualizations of Graph Structures

To provide further details on the experimental setup, we present the specific network topologies used in our evaluation. Figure 7 illustrates the eight distinct graph structures employed. These include both structured synthetic grids and irregular real-world road networks.

C Visualization Evaluation

This section presents the Visualization evaluation method provided by GraphChase. Figure 8 captures snapshots from a recorded pursuit-evasion episode, illustrating the position changes of the pursuers and the evader at each time step. Step 1 shows the initial positions of the evader and the pursuers. Each subsequent step reflects one move by each side. In Step 6, the evader and pursuer overlap, indicating that the evader has been captured by a pursuer.

D Computational Scalability Analysis

As discussed in the main text, current methods often rely on the exhaustive enumeration of the evader's action space. This requires pre-calculating all valid simple paths from the starting position to

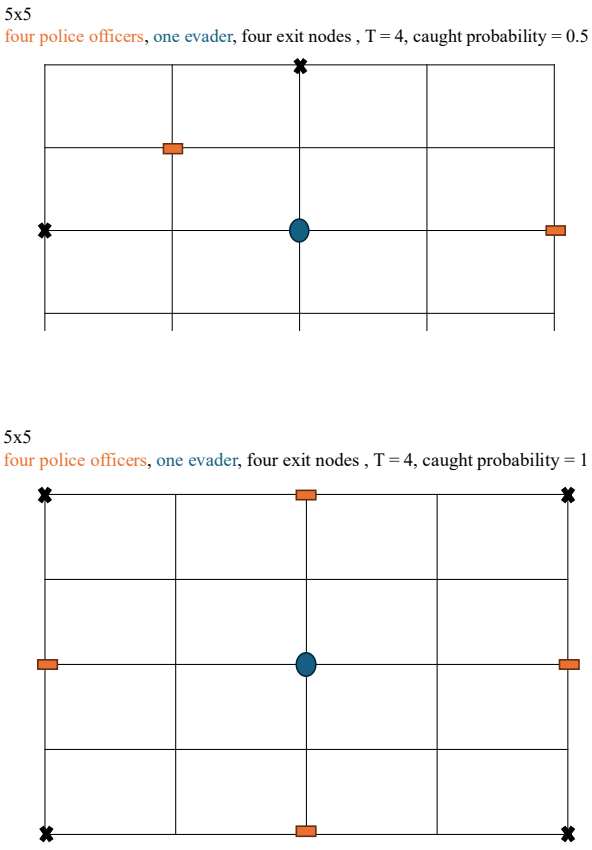

Figure 6: 5×5 grid graph settings with the evader caught probability of 0.5 (left) or 1 (right).

the exits within a finite time horizon T. While feasible for small maps, this approach faces severe scalability issues.

To quantify this limitation, we measured the time cost required to enumerate simple paths from the center of a grid (N×N) to the corners, setting the time horizon $T = N$. Figure 9 illustrates the relationship between grid size and computation time. The results demonstrate an exponential explosion in computational overhead. Specifically, the runtime becomes prohibitively high once the grid size exceeds 20×20. This trend confirms that reliance on global action space traversal constitutes a fundamental scalability barrier, necessitating the data-driven approach proposed in this work.

E User-controllable parameters

The user-controllable parameters are shown in Table 5. In our platform, users configure a graph-based pursuit-evasion game by providing a **base graph structure** and specifying game-specific settings through the `GameSettings` class. The base graph can be loaded from a NetworkX `gpickle` file, which defines the topology of the environment. We also provide a graph generation utility that allows users to create grid-based graphs through parameters such as `grid_width` and `grid_height`, which define the grid dimensions;

Table 5: The parameters that users can control.

	Graph Generation	Graph Customization
underlying graph structure	grid_width grid_height axis_edge_prob diag_edge_prob	adjacency_matrix node_ids use_weighted_graph
agent number and position	time_horizon evader_init pursuer_init exit_nodes graph_gpickle_path	

`axis_edge_prob`, which controls the probability of retaining horizontal and vertical edges; and `diag_edge_prob`, which determines the probability of adding diagonal edges. These parameters enable users to generate diverse graph topologies ranging from sparse to fully-connected grid structures.

Users can further customize the graph structure through the `metadata` parameter. By specifying an `adjacency_matrix` along with corresponding `node_ids`, users can override the base graph topology with a completely custom structure. The adjacency matrix can be either **symmetric** (for undirected graphs) or **asymmetric** (for directed graphs), with each entry representing the edge weight between nodes. This flexible design enables users to model diverse scenarios, from simple grid-like environments to complex directed weighted networks. Additionally, the `use_weighted_graph` parameter controls whether edge weights are utilized in the game dynamics; when set to `False`, all edges default to unit weight.

For the **agent configuration and game parameters**, users control the number and initial positions of agents through `evader_init` and `pursuer_init`, which specify the starting node indices for each type of agent. The `exit_nodes` parameter defines the set of goal locations that evaders aim to reach, while `time_horizon` sets the maximum simulation duration. This parameterization allows users to construct a wide range of pursuit-evasion scenarios tailored to specific research questions or applications.

F Faster wall-clock Convergence

Our platform incorporates several architectural and implementation enhancements that contribute to faster wall-clock convergence times compared to the original implementations. These improvements span environment simulation, configuration management, data processing pipelines, and computational resource utilization.

Standardized Environment Interface. We have adopted the Gymnasium framework as the standard interface for all game environments, replacing the custom class-based implementations found in the original papers. The original code (e.g., `build_game` in `game_config.py`) constructs game instances through deeply nested function calls with numerous embedded parameters, resulting in redundant data copying and inefficient state transitions. In contrast, our `UNSGEnv` class implements the standard Gymnasium API with optimized step and reset methods, eliminating unnecessary data transformations between the environment and agents. This standardization not only accelerates simulation processes but

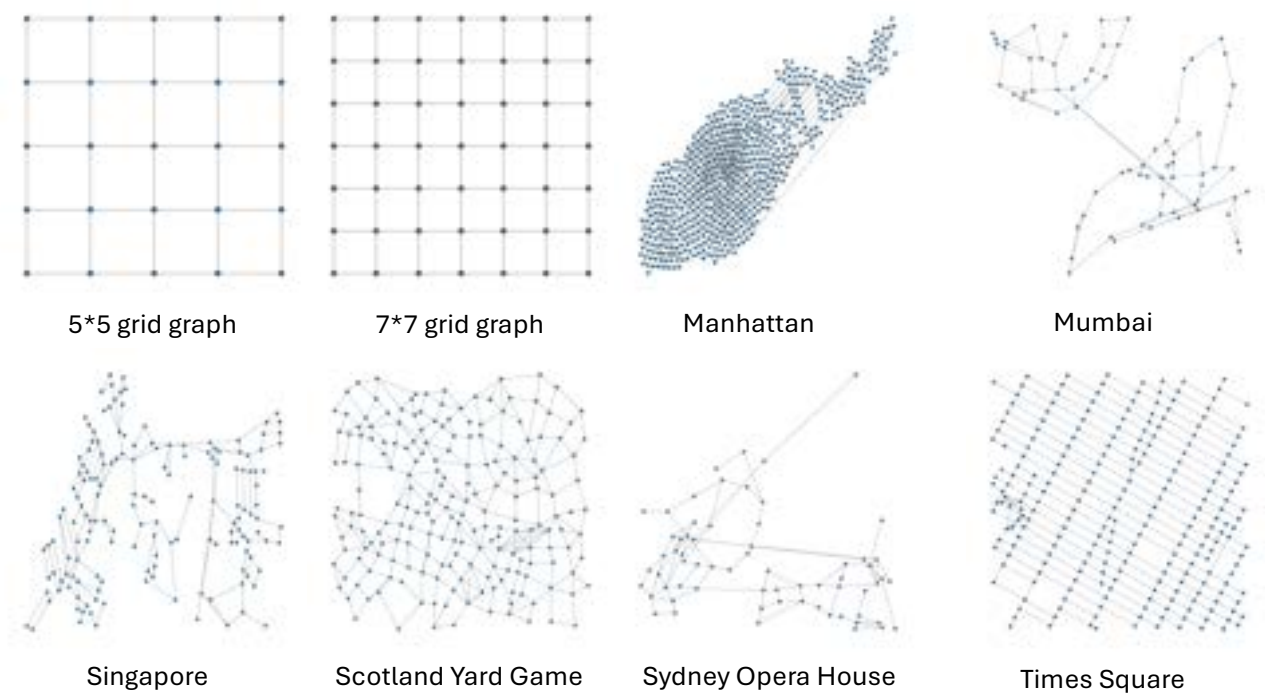

Figure 7: Visualization of the eight graph structures employed in the experiments. The dataset comprises synthetic grid graphs ($5 \times 5, 7 \times 7$) and real-world topologies.

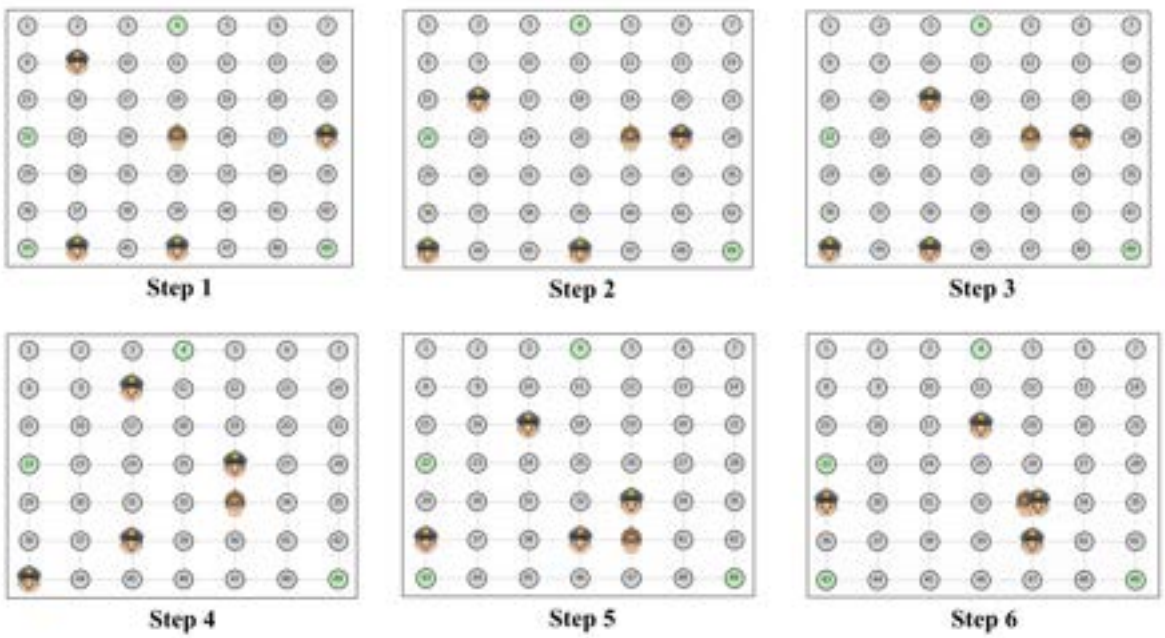

Figure 8: Snapshots from a recorded pursuit-evasion episode.

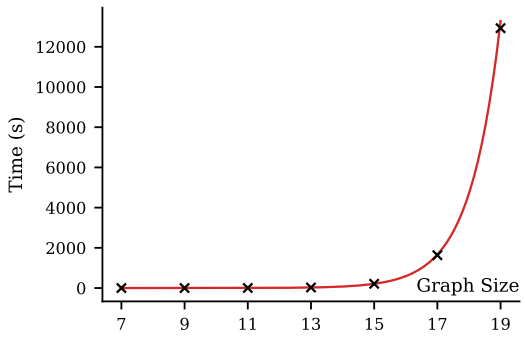

Figure 9: Time cost for enumerating evader simple paths from the grid center to four corners. The solid line represents the exponential trend, fitted via linear regression on the logarithmic scale. The rapid growth underscores the intractability of path-based methods on large-scale graphs.

also ensures compatibility with modern reinforcement learning libraries.

Efficient Graph Representation and Loading. The original implementation rebuilds graph structures during environment initialization, often involving multiple graph transformations and node embedding computations for each experiment run. Our platform leverages NetworkX's `gpickle` serialization format to precompute and cache graph structures. Graphs are loaded once via `pickle.load()` and wrapped in a `GameSettings` object, which efficiently manages edge weights and neighbor mappings through pre-built dictionaries. This approach eliminates redundant graph construction overhead across training iterations.

Streamlined Configuration Management. We employ YAML-based configuration files combined with command-line overrides (via `load_yaml_config`), replacing the hardcoded parameter specifications scattered throughout the original codebase. The original implementation embeds configuration parameters directly in execution scripts, making parameter tuning and reproducibility challenging. Our configuration system centralizes all hyperparameters, enables programmatic parameter sweeps, and automatically saves experiment configurations via `save_experiment_config`, facilitating efficient experimentation and result tracking.

Optimized Data Processing Pipeline. We have implemented various data processing optimizations throughout the training pipeline. The original code frequently converts between Python lists, numpy arrays, and PyTorch tensors, incurring significant overhead during high-frequency operations such as state observation extraction and action sampling. Our implementation maintains consistent data representations using numpy arrays for observations (via `get_current_obs` returning structured numpy arrays) and PyTorch tensors for neural network computations, minimizing type conversion costs. Additionally, we have optimized the dimension computation functions (`compute_input_dim`, `compute_action_dim`) to perform calculations once during initialization rather than repeatedly during training.

Modular Component Design. Our platform separates concerns through well-defined abstractions: `env_builder` for environment instantiation, `PPOAgent` for policy networks, `PPOAlgorithm` for learning updates, and specialized runners (`AttackerPathRunner`, `DefenderPretrainPsroRunner`) for training orchestration. This modularity contrasts with the monolithic `PSRO_Oracle` class in the original implementation, which tightly couples environment interactions, policy updates, and meta-strategy computation. Our design reduces redundant computations by enabling component-level caching and reuse across PSRO iterations.

Vectorized Environment Support. Unlike the original sequential episode sampling, our platform supports vectorized environments through the `vec_envs` parameter, enabling parallel data collection across multiple environment instances. This parallelization capability substantially reduces wall-clock training time, particularly during the pretrain phase where large numbers of episodes (`pretrain_episodes_per_task`) are required.

It is important to emphasize that these enhancements primarily accelerate wall-clock convergence without altering the fundamental algorithmic behavior. The number of training iterations required for convergence remains approximately consistent with the original implementations. For instance, if the original PSRO algorithm requires 10^4 episode samples to converge, our platform's reproduced version similarly requires a comparable number of training iterations. This consistency validates that our optimizations preserve algorithmic fidelity while delivering substantial wall-clock speedups through improved implementation efficiency.

G Usage Instructions for GraphChase

The following steps outline the process for setting up and utilizing the GraphChase platform:

G.1 Cloning the Repository

To begin, clone the GraphChase repository from GitHub and navigate to the project directory:

```
git clone \
https://github.com/GraphChase/GraphChasePlatform.git
cd GraphChasePlatform
```

G.2 Installing Dependencies

Install the necessary dependencies, including PyTorch, NetworkX, Gymnasium, and other required packages. Refer to the requiremen ts.txt file for the complete list of dependencies.

G.3 Preparing Graph Structures

Our platform uses NetworkX `gpickle` format for graph storage. Pre-generated graph files are available in the new_version/graph/ custom_graph/ directory. To create custom graphs, use the graph generation utility:

```
python -m new_version.graph.generate_custom_graph \
  --grid_width 7 --grid_height 7 \
  --axis_edge_prob 1.0 --diag_edge_prob 0.0 \
  --output_path custom_graph.gpickle
```

G.4 Running an Algorithm

To run a specific algorithm, such as Pretrain-PSRO, perform the following steps:

(1) **Select or Create Configuration File:** Choose an appropriate YAML configuration file from the `new_version/solver_cfgs/` directory (e.g., `pretrain_psro_cfgs_7_7_5.yaml`), or create a new one based on existing templates. The configuration file specifies:
   - **Graph arguments:** Path to the gpickle file (graph_gpickle_path), agent initial positions (attacker_init, defender_init), exit nodes (`exit_nodes`), and time horizon
   - **Graph embedding arguments:** Whether to use graph embeddings, embedding parameters, and paths to pre-computed embeddings
   - **Algorithm hyperparameters:** PSRO iterations, training batch sizes, PPO learning rates, and other algorithm-specific parameters
   - **Experiment settings:** Random seed, GPU configuration, output paths, and logging preferences

(2) **Customize Parameters (Optional):** To override specific configuration values without modifying the YAML file, use command-line arguments with the `–set` flag. For example, to change the random seed:

```
--set seed=3407
```

(3) **Run the Algorithm:** Execute the algorithm using one of the following methods:

Method 1 - Direct Python Execution:

```
python -m new_version.scripts.run_pretrain_psro \
    --config new_version/solver_cfgs/\
pretrain_psro_cfgs_7_7_5.yaml \
    --set seed=3407
```

Method 2 - Shell Script for Multi-seed Experiments:

```
bash run_pretrainpsro_multiseed.sh
```

This script automates the process of running experiments across multiple random seeds, particularly useful for statistical validation and cluster computing environments.

All algorithms utilize the same YAML-based configuration system, enabling consistent parameterization and reproducible experimentation across different solution methods.

H Hyperparameter Configuration

The hyperparameter configurations for all implemented algorithms are detailed in Tables 6, 7, 8, 9, and 10. These configurations are based on the experimental setting with a 7×7 grid graph. For additional experimental configurations, including different graph sizes, agent numbers, and real-world map scenarios, pre-configured YAML files are available in the `new_version/solver_cfgs/` directory of our GitHub repository.

Table 6: Hyperparameters for Pretrain-PSRO algorithm (7×7 grid, captured $prob = 0.5$).

Game Configuration			
time_horizon	7	seed	77
use_weighted_graph	false	device_id	0
Graph Embedding Parameters			
graph_embeddings	true	emb_size	16
node_information_type	all	normalize_info	true
similarity	cosine	line_order	all
epochs	200	batch_size	32
neg_samples	5	lr	0.025
PSRO Parameters			
num_psro_iteration	20	eval_episodes	1000
rollouts_per_attacker_action	10000	vec_envs	16
train_defender_batches	630	episodes_per_batch	8
Pretraining Parameters			
pretrain_iterations	40	pretrain_tasks	30
pretrain_episodes_per_task	20	load_pretrained_model	false
PPO Hyperparameters			
ppo_actor_lr	0.001	ppo_critic_lr	0.003
ppo_gamma	0.99	ppo_lambda	0.95
ppo_clip	0.2	ppo_epochs	10
ppo_batch_size	32	ppo_hidden_dim	128
entropy_coef	0.01		
Attacker Settings			
action_type	exit_node	strategy_type	mix
attacker_path_type	shortest	reward_mode	win_rate

Table 7: Hyperparameters for CFR-Mix algorithm (7×7 grid, captured $prob = 0.5$).

Game Configuration			
time_horizon	7	seed	77
use_weighted_graph	false	device_id	0
Network Architecture			
network_dim	32		
Training Parameters			
iteration	10000	train_epoch	1000
sample_number	20	action_number	1000
Attacker Regret Network			
attacker_regret_batch_size	32	attacker_regret_lr	0.0015
Defender Regret Network			
defender_regret_batch_size	512	defender_regret_lr	0.0015
Defender Strategy Network			
defender_strategy_batch_size	32	defender_strategy_lr	0.0015

Table 8: Hyperparameters for NSGZero algorithm (7×7 grid, captured $prob = 0.5$).

Game Configuration			
time_horizon	7	seed	77
use_weighted_graph	false	device_id	0
save_model	true	use_tensorboard	false
Network Architecture			
embedding_dim	16	hidden_dim	256
Training Parameters			
max_episodes	6000	batch_size	128
buffer_size	50000	lr	0.0005
num_workers	4	debug_single_process	false
Training Schedule			
train_every	16	train_from	128
test_every	250	test_nepisodes	50
save_every	500	log_every	500
MCTS Parameters			
num_sims	150	bias	0.5
cpuct	0.3	temp	0.5
gamma	1.0	reward_mode	win_rate
Attacker Configuration			
att_type	nfsp	ban_capacity	500
cache_capacity	20	br_rate	0.2

Table 9: Hyperparameters for NSG-NFSP algorithm (7×7 grid, captured $prob = 0.5$).

Game Configuration			
time_horizon	7	seed	0
use_weighted_graph	false	device_id	0
Network Architecture			
embedding_size	32	hidden_size	64
relevant_v_size	64	if_naivedrrn	false
seq_mode	cnn		
Replay Buffer Configuration			
br_buffer_capacity	500000	avg_buffer_capacity	10000000
Learning Parameters			
br_lr	0.0001	avg_lr	0.0001
br_batch_size	128	avg_batch_size	256
Exploration Parameters			
d_expl	0.0	a_expl	0.1
br_prob	0.1		
Training Schedule			
max_episodes	1200000	train_br_freq	4
train_avg_freq	32	check_freq	100000
check_from	100000	display_freq	10000
min_to_train	1000	exact_br	false
Agent Configuration			
defender_rl_mode	drrn	defender_sl_mode	drrn
attacker_mode	bandit	reward_mode	win_rate

Table 10: Hyperparameters for GRASPER-MAPPO algorithm (7×7 grid, captured $prob = 0.5$).

Game Configuration			
time_horizon	7	seed	77
use_weighted_graph	false	device_id	0
graph_type	Grid_Graph	edge_probability	0.8
row	7	column	7
Graph Embedding Configuration			
use_node_emb	true	load_graph_emb_model	true
max_epoch	200		
Game Pool Configuration			
load_game_pool_file	true	pool_size	1000
min_num_defender	4	max_num_defender	4
min_num_exit	4	max_num_exit	5
min_attacker_pth_len	5		
PSRO Parameters			
num_psro_iteration	20	train_defender_batches	1200
episodes_per_batch	32	reward_mode	win_rate
MAPPO Hyperparameters			
ppo_epoch	8	minibatch_size	64
use_gae	false	use_act_supervisor	true
Training Schedule			
num_iterations	20000	save_every	1000