MSc Artificial Intelligence
Master Thesis

Predicting plausible escape routes using reinforcement learning and graph representations

by
Rogier van der Weerd MSc
Student nr. 13454242

July 15, 2022

Thesis work (48EC)
November 2021 - July 2022

Supervisor:
D. Kuric MSc (UvA)
Mentors:
I.S. Van Droffelaar MSc (TU Delft)
T. Kastelein MSc (Dutch Police)

Examiner:
Dr. H.C. Van Hoof (UvA)

Universiteit van Amsterdam

Contents

Abstract

A search & pursuit strategy by police units (pursuers) is aimed at capturing a suspect (the escaper) that attempts to flee a crime scene and evade interception. This problem can be modeled using graph traversal, where the graph represents a road network with intersections (vertices) and roads (edges). Many modeling choices are involved in order to represent this real-life problem in a meaningful and feasible way. Research at the Dutch Police has focused on finding pursuer strategies that maximize interception probabilities using various optimization techniques. However, the quality of these strategies is difficult to assess given the uncertainty of how an escaper will behave while on the run. One way to assess a pursuer strategy is to test whether it can be exploited by an escape agent. This requires that the escaper can learn a general strategy that is effective on any instance of any road network.

This thesis work investigates whether Reinforcement Learning in combination with Graph Representation Learning can be applied to find escape policies that exploit weaknesses in these pursuer strategies. Ideally, escape policies can be learned for any (unknown) pursuer strategy and such escape policies generalize over different (unseen) graphs and different initial conditions (such as number of pursuers and their positions at the start of an episode). This represents a challenging task given that i) the escape agent operates under partial observability because it does not always observe all pursuers, ii) the problem is NP-hard with a combinatorial search space that explodes with increasing graph size and complexity, and iii) there are infinitely many combinations of graph topologies and initial positions of the agents.

We show that this task can be learned by defining an escape agent model as a parametrized value or policy function. These models consist of a graph neural network to extract latent node embeddings, combined with an LSTM module to account for previously encountered graph states. The agent models are trained using Deep-Q Learning (DQN) and Proximal Policy Optimization (PPO), respectively. Such trained escape agents can learn to exploit a given pursuer strategy and outperform a collision risk avoidance heuristic on unseen graphs, over a range of observability levels and for an arbitrary number of pursuers. This approach can be used to define a performance metric of pursuer strategies and expose vulnerabilities in such strategies.

Acknowledgements

This thesis project was performed under the academic internship program of the ICAI Police Lab AI, hosted by Team Wetenschappelijke Ontwikkelingen (TWO) of the Dutch Police in the period November 2021-July 2022. I would like to thank Theo van der Plas, Marius Kok and Bas Testerink for offering this opportunity and for taking an interest in the work, as well as all TWO team members for creating a welcoming and supportive atmosphere. I enjoyed the interactions and great discussions.

A special thank you to my external and internal supervisors for providing invaluable advice, guidance and inspiration: Irene van Droffelaar (TU Delft) and Tamar Kastelein (Dutch Police), whose work formed the basis of this research; Judith Klepper (Dutch Police) who made sure everything ran smoothly; David Kuric (University of Amsterdam / AMLab) who helped me navigate the various challenges that come with experimental Reinforcement Learning. I'm grateful to have had the opportunity to join a full surveillance shift led by Wout Bouvé (Dutch Police, Dienst Infrastructuur) and to be able to interact with the officers on duty. Seeing seasoned professionals in action and experiencing on-the-ground realities informed my attempt to strike a balance between theory and practice - a crucial aspect of effective and responsible use of technology in support of human decision making.

Whilst making known my appreciation of the people who have helped me with advice or comment, I stress that the responsibility of any errors or omissions is mine alone.

Chapter 1

Introduction

1.1 Search and pursuit in law enforcement

The Dutch Police are interested in decision support applications that can improve performance in search & pursuit scenarios of suspects fleeing a crime scene. In such scenarios, deployed police units are guided by operators in a control room that use real-time centralized information gathering and coordination. Currently, decision making is based on experience and intuition by both dispatchers and field units given limited and noisy information about the situation on the ground. Effective response can be hindered by many factors, including lack of information, sub-optimal use of data, miscommunication and non-compliance of instructions. There could be value in a system that suggests best actions for search pursuit units (for instance, where to direct them) given all available information.

Kastelein [Kas20] developed such a system by using a graph-based representation of a road network within a bounded area around the crime-scene. She frames the optimization as a satisfiability problem in order to output pursuer routings that maximize the number of potential escape routes that are intercepted within a certain number of time-steps. It assumes random behavior by the evader, possibly with directional preference. Her work was used to develop a prototype system called IMA[1] that is able to apply the optimization on the Dutch road network, as one step towards bringing such information into the actual police operation. Based on the initial position of an escaper, IMA predicts a number of potential escape routes and calculates the optimal positioning of interception units, given their initial positions. Van Droffelaar [vD21] is continuing this research with the purpose of improving the decision support method and identifying timely, robust and explainable fugitive interceptions, given limited real-time data. It is an open question how to compare and rank different methods based on their effectiveness.

Understanding performance and robustness of these search & pursuit optimization methods using an objective and reliable metric is crucial given the stakes in any real-life deployment. However, such performance measurement is challenging as escape behaviour is unpredictable and hard to model. Real-life data of escape behavior is difficult to acquire and can only be observational, not interventional. One avenue worth investigating is the use of Reinforcement Learning (RL) to train an escape agent, based on experience acquired in a simulated environment where the pursuer strategy is applied. Such a trained agent may expose weaknesses by learning to exploit these pursuer strategies. This information may be used to evaluate, and possibly improve the performance of pursuer strategies. RL has a number of advantages over other potential methods (such as heuristics, probabilistic search, etc.), the most important being that an RL agent can learn from interactions without knowing anything about the pursuer's model a priori.

Note that RL could be used to develop pursuer strategies as well, potentially in a setting of adversarial learning. However, the primary goal of this work is to learn escape policies in order to assess the performance of pursuer strategies.

[1] Intelligente Meldkamer Assistent

1.2 Problem definition

The domain of Search & Pursuit-Evasion (SPE) problems constitutes a large family of problems [FT08, Als04, CHI11], with many variants depending on aspects such as the number and type of agents (mode of transport, capability[2]), their degrees of freedom in movement (the action space), their objectives, level of observability[3], level of cooperation, type of environment, etc. Many modeling decisions and assumptions are made in order to define and evaluate escape/pursuer behaviors. Our goal is to create a model that adequately represents real-life scenarios, while minimizing its complexity to ensure computation and optimization remains feasible. As a starting point, two key aspects of the problem can be defined.

Task and campaigns

In the context of search & pursuit optimization for the purpose of this study, an escape agent with a certain objective is located on a road network, and any number of pursuers are present on the network with the aim of capturing the escaper using coordination. The escaper's objective might simply be not to get caught, possibly combined with a goal to reach a target location, or to increase its distance to the crime scene. We distinguish:

- *Search campaigns*: the exact escaper position is unknown and there is no current visual identification of the escaper. There may be knowledge of recent escaper positions. The primary goal of pursuit units is to locate the escaper. Search campaigns can be:
  - Static: deployed pursuers assume strategic stationary positions with the objective of visual identification of an escaper
  - Dynamic: deployed pursuers navigate strategic surveillance routes with the objective of visual identification of an escaper
  - Hybrid: deployment of a mix of passive and dynamic search agents

  In practice, search campaigns will be hybrid as agents will (rightfully) apply judgement to decide between holding a position and moving around. However, the distinction will be useful when modeling search & pursuit behaviors.
- *Pursuit campaigns*: current escaper location and direction is available through active visual contact by at least one pursuer. All pursuers coordinate with the objective of intercepting and stopping the escaper.

These two modes are different in nature: finding a target requires a different approach than stopping and capturing a target that is in sight. The primary interest of the Police lies in the effectiveness of search campaigns as the potential for improvement of this task is expected to be highest. The scope of this thesis is limited to static search campaigns.

Problem setting

The search & pursuit problem can most naturally be defined on a graph that is representative of the road network. This representation can be literal, with nodes and edges representing physical intersections and road segments. The representation may also contain some level of abstraction, for instance by considering nodes to represent a general geographic area that can be observed when a pursuit agent is located anywhere in that area, and edges representing (coarsened) ways to move from one area to another. Various graph classes, ranging from small example graphs to large and realistic graphs. A graph state is defined by the set of node positions of all agents at a specific time step, combined the static set of target nodes for the escaper. The events of visual identification or capture can be defined based on this graph state (i.e. escaper and at least one pursuer move to the same node), or transitions between states (i.e. their walks have crossed on an undirected edge during a transition). Fig.A.1 in Appendix A illustrates this basic setting.

[2]E.g., a helicopter or road camera can only observe, a police car can observe and intercept, etc.

[3]Which information (location, direction/speed of travel, intentions) on other agents is known at which point in time

1.3 Approach and contributions

In this work, we apply a combination of Graph Representation Learning and Reinforcement Learning (RL). This combination has been shown to be powerful in a range of combinatorial optimization problems [CCK+21], including routing problems on graphs [KDZ+17, KvHW19, VFJ15]. In order to solve the Search & Pursuit-Evasion problem introduced above, we define an escape agent model as a parametrized value- or policy function approximation. These models consist of a graph neural network to extract latent node embeddings, combined with an LSTM module [HS97] to account for previously encountered graph states. The agent models are trained using Deep-Q Learning (DQN [MKS+13]) and Proximal Policy Optimization (PPO [SWD+17]), respectively. We will that such trained escape agent models can outperform a collision risk avoidance heuristic on unseen graphs, over a range of observability levels and for an arbitrary number of pursuers. This approach can be used to define a performance metric of pursuer strategies and expose vulnerabilities in such strategies.

A number of contributions are made:

- Our approach is an example of how RL can be used to validate and evaluate optimizations that originate from classical methods used in Operations Research (OR). Such applications are rare in RL/OR literature.
- A specific Search & Pursuit-Evasion scenario, relevant in the context of police operations, is formalized as a Partially Observable Markov Decision Process (POMDP). This enables the application of RL methods to solve the SPE task from the perspective of the escape agent.
- Existing methods (DQN and PPO algorithms, Graph Neural Networks that yield node- and graph state embeddings, Recurrent Neural Networks for state reconstruction) are integrated in order to solve escape optimization on graphs under partial observability. The combination of these techniques is not common, and modern RL frameworks do not offer the flexibility in architectures and training regimes required for our experiments. Hence a custom environment and implementation of these algorithms is developed and made available (Appendix D).
- A new dataset is created that contains a wide range of graph classes and graph instances (defined by the number of pursuit agents and the initial positions of escape- and pursuit agents). Pursuer responses under different pursuer strategies are pre-calculated in order to enable RL algorithms to train efficiently. The dataset will be made public, pending approval by the Dutch Police.
- Using the above-mentioned techniques, an evaluation metric is suggested that can be used to rank different pursuer strategies. This ranking is based the performance and robustness of a pursuer strategy against a sophisticated escape agent model that is trained on a pre-defined trainset with a certain computation budget.

The aim of this work is to show the potential and limitations of RL in this application domain. Additionally, other areas for further research and improvement of the method are identified. The translation of the methods to reliable and safe deployment in actual police operations is left for future work.

Chapter 2

Research Question

Given the problem definition described above, we consider a setting of an escape agent that traverses a directed graph. The graph contains a certain number of pursuit agents that are engaged in a search campaign using a strategy unknown to the escaper. The escaper has a primary objective of not being found and (potentially) a secondary objective of reaching a sub-set of nodes on the graph (considered a safe haven) with some level of urgency.

The main research question of this work is as follows:

> How can Reinforcement Learning (RL) be applied to find an optimal strategy to escape from pursuer adversaries on a graph, such that this strategy generalizes to i) any set of initial positions, ii) any number of pursuers, iii) any (unseen) graph, iv) any level of observability?

We limit the work to involve pursuers that engage in search campaigns and assume their behavior to be a black box (the IMA system, unknown to the escape agent) that can be interrogated. The escape agent is trained with some computation budget (number of trial episodes) and its performance is measured on graphs and graph instances it has not observed before (zero-shot learning). Several aspects of this problem need to be addressed:

- How to model the escape task such that it captures sufficiently realistic real-life behavior and constraints so that it can be applied to actual police operations, while it remains feasible to perform optimization on it?
- How to ensure that an escape strategy can generalize to the four dimensions described in the research question?
- If successful and generalizable escape strategies can be found, how can this knowledge be used define a performance metric of pursuer strategies?

Our hypothesis is that a combination of Graph Representation Learning (using graph neural networks) and Reinforcement Learning can be effective in this application. Recent advances in these techniques demonstrate promise in the ability to create latent representations of graph states and learn the abstract heuristics required to solve these type of combinatorial optimization problems on graphs.

Questions that are out of scope for this work include:

- How can RL be used to learn pursuer strategies, possibly in an adversarial setting?
- How to translate findings into a decision support model that can be deployed in actual police operations?
- How to extend this work to pursuers that engage in active pursuit?

Chapter 3

Related Work

This work looks to bring to together three distinguishiable areas of research: i) Multi-agent Search and Pursuit-Evasion (SPE) problems, ii) Reinforcement Learning and iii) Graph Representation Learning. The first two of these already have a range of combined applications. However, applying geometric deep learning to the SPE domain has not been extensively studied to date, although it is a natural extension of substantial work already done in applying GNNs to solve combinatorial / NP-hard graph-based problems.

3.1 Search and Pursuit-Evasion (SPE) problems

The starting point of this thesis work is formed by [Kas20], who translated the pursuit optimization problem in the scope of decision support in a control room of the Dutch Police. She applies Integer Linear Programming to output pursuer routings that maximize the number of potential escape routes that are intercepted within a certain number of time-steps. [vD21] builds on this work with the aim of improving performance and explainability of the solutions, which will help in adoption by practitioners.

Research on SPE problems goes back to early game theoretic formulations [ICfIM65] and formulations on graphs [Par78], where solutions under full information and alternating turns (dubbed *Cops and Robbers games*) are offered. [Als04, CHI11] survey and categorize many variants of the problem and classical approaches to solving them such as the marking algorithm, probabilistic search and other methods developed in operations research. Small and specific formulations may have exact solutions, such as [WW95] that uses network flow techniques. Given the computational complexity, however, most existing solutions are heuristic in nature and few approximation algorithms have provable performance guarantees. If performance guarantees exist, they tend to apply only on very small graph instances.

3.2 (Deep) Reinfocement Learning in the SPE domain

Many recent successes that combine deep learning with reinforcement learning techiques have shown their combined potential in intelligent decision making and control [BRW$^+$19]. These techniques have also been introduced in the SPE domain with the main advantage they do not require differential game models: agents learn optimal strategies solely by interacting with the environment. Most approaches aim to solve cooperative multi-agent tasks from the perspective of the pursuers with a range of methods, using (deep) Q-learning (DQN) [BKU15, GEK17], various policy based methods such as Trust Region Policy Optimization (TRPO) [HAN$^+$19], Deep Deterministic Policy Gradient (DDPG) [LWT$^+$17], and extensions such as curriculum learning [dSNC$^+$21]. [LWT$^+$17] extends DDPG into a multi-agent policy gradient algorithm where decentralized agents learn a centralized critic based on the observations and actions of all agents.

Particular challenges are faced when addressing limited observability by the agents, typically described by Partially Observable Markov Decision Processes (POMDPs). There is well developed

research into dealing with POMDPs in RL such as Deep Recurrent Q-Learning (DRQN) [HS15] (dealing with flickering screens in DQN for Atari games by jointly training convolutional and LSTM layers), [ZLPM17](incorporating past actions in state histories), [SSP+15] (adding attention layers to DRQN), and others [SYML18, ZLPM17], mostly by introducing recurrent neural networks in the parametrized value or policy models. This idea was first shown to be feasible by [WFPS07], who used LSTMs [HS97] to map histories (memory states) to action probabilities in a Policy Gradient framework. Stability and convergence issues need to be addressed when using RNNs in combination with experience replay, investigated by [KOQ+19, SYML18]

[OAMAH15] extends the notion of decentralized POMDPs for multi-agent cooperation with asynchronous decision making, improving scalability by using belief space macro actions in planning. The challenge of generalizing policies such that they can be applied to different number of agents than what they were trained on was attempted by [XHG+19] using bi-directional LSTMs with DDPG.

3.3 Graph-based Reinforcement Learning

The field of Graph Representation Learning has gained importance given its ability to use inherent structure of information (encoded in the graph) in machine learning algorithms [Ham20]. This approach has seen breakthroughs in tasks such as node classification ([KW17] who introduced graph convolutions), relation/edge prediction (survey by [NMTG16]) and graph-level classification/regression/clustering (e.g. [GSR+17] who formulated message passing neural networks and applied it to molecular property prediction). [Bea18] and [GNNSys21[1]] provide excellent overviews of the state of this field.

Particularly relevant for SPE problems are recent insights into combinatorial optimization and reasoning using graph neural networks and RL, surveyed in [CCK+21, PCX21]. The SPE task can be framed as a routing problem, where next nodes need to be selected based on a graph state (which may include a node selection history). Early ideas involved *Pointer Networks* [VFJ15, BPL+16, VBO+20] where a recurrent network is trained in a supervised manner to predict the sequence of nodes (for instance, visited cities in the Travelling Salesman Problem). [KDZ+17] approaches the same problem using *structure2vec* [DDS16], a structured data representation built by embedding latent variable models into feature spaces, and learning such feature spaces using discriminative information from graphs. [KvHW19] successfully applies *Graph Attention Networks* [VCC+18] to learn heuristics to solve routing problems on graphs.

It is expected that these methods can be applied to the SPE formulation studied in this thesis. This requires an appropriate definition of edge- and node features and a network architecture that is able to deal with unknown and potentially stochastic behavior of the pursuer adversaries. Promising research [XLZ+20, VYP+20] investigates how network architecture can be matched with the nature of the optimization problem (*algorithmic alignment*) in order to enable generalization. The conditions under which GNNs are performant and generalize well is still an open question, with various recent contributions [PY21, XHLJ19, KTA19]. Lastly, any approach should be scalable to reasonably sized road networks (>1k node graphs). [MMD+19] introduces scalability techniques using a probabilistic greedy mechanism to predict the quality of a node, which may provide useful in our application.

[1] https://gnnsys.github.io/

Chapter 4

Background

We introduce the core concepts and notation that we will be using in this thesis, starting with two Reinforcement Learning algorithms: Deep Q-Learning (DQN) and Proximal Policy Optimization (PPO). These algorithms are applied in a partial observability setting. We make use of recurrent neural networks to estimate the full state in such a setting. Lastly, two approaches to graph representation learning are introduced: *structure2vec* and Graph Attention Networks (*GAT*).

4.1 Reinforcement learning

Reinforcement Learning can be defined as 'the study of how an agent can interact with its environment to learn a policy which maximizes expected cumulative rewards for a task' [SB18]. RL can be considered as a branch of AI, positioned at the center of the 'science of decision making'[1] with many connections to related fields including engineering (dynamical systems and control), mathematics (operations research), computer science, economics, neuroscience and psychology. Learning occurs by accumulating experience: the agent explores a real-life or simulated environment (with dynamics that are not necessarily known by the agent a priori) and receives some reward signal. The key of all RL algorithms is to derive a behavioral policy based on a history of actions taken and subsequent rewards received over many episodes. RL has distinct characteristics: there is no supervisor, only a reward signal. Feedback on actions taken is delayed, raising an attribution challenge to determine the value of individual actions. Decision making is sequential and an agent's actions affect subsequent data it receives which makes the data inherently non-i.i.d. (not independently and identically distributed).

Maximizing returns Our environment can be modeled as a finite and discrete[2] Markov Decision Process (MDP) $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}(\boldsymbol{s}), p_0, P, R, \gamma \rangle$ with $\boldsymbol{s} \in \mathcal{S}, \boldsymbol{a} \in \mathcal{A}(s)$ and $\mathcal{S}, \mathcal{A}(\boldsymbol{s})$ finite sets of discrete possible states and actions, p_0 a distribution from which an initial state $\boldsymbol{s}_0 \sim p_0$ is drawn, $P : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ the state transition probability, $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ a reward function and $\gamma \in (0, 1]$ a discount factor used to define a present value of future expected rewards. An agent acts according to some policy $\pi(\boldsymbol{a}_t|\boldsymbol{s}_t) : \mathcal{S} \times \mathcal{A} \to [0, 1]$ that defines a probability distribution from which actions $\boldsymbol{a}_t$ are sampled given a current state at each timestep. The state $\boldsymbol{s}_t$ is said to have the Markov property if it includes information on all aspects of the past agent-environment interaction that make a difference for the future. In other words: $\boldsymbol{a}_t$ and $\boldsymbol{s}_t$ fully determine the probability of each possible value of $\boldsymbol{s}_{t+1}$ and r_{t+1}. By sequentially acting according to $\pi(\boldsymbol{a}|\boldsymbol{s})$ a trajectory $\tau = (\boldsymbol{s}_0, \boldsymbol{a}_0, r_1, \boldsymbol{s}_1, \boldsymbol{a}_1, r_2, ..., \boldsymbol{s}_T, \boldsymbol{a}_T, r_{T+1})$ is formed that ends with a final reward in an end state at $t = T + 1$ when some terminal condition is satisfied. The discounted

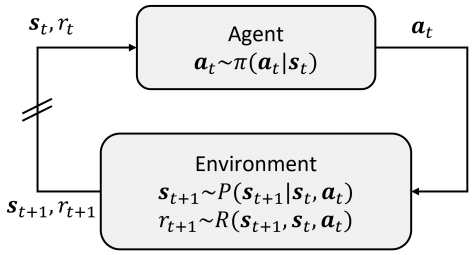


Figure 4.1: Agent-environment interaction in an MDP (adapted from [SB18]

[1] https://www.davidsilver.uk/teaching/

[2] Reinforcement Learning theory extends to non-episodic and continuous time tasks but we will not require it

return of a trajectory at timestep t can be defined as $G_t(\tau) = \sum_{k=t}^{T} \gamma^{k-t} r_{k+1} = r_{t+1} + \gamma G_{t+1}(\tau)$. The goal in RL is to find an optimal policy $\pi^*(\boldsymbol{a}|\boldsymbol{s})$ that maximizes the expected discounted return over full episodes $J = \mathbb{E}_{\tau\sim\pi}[G_0(\tau)] = \mathbb{E}_{\pi}[G(\tau)]$. Two value functions are defined that describe this objective. The state-action value function, or *Q-function*, $q_\pi(\boldsymbol{s}, \boldsymbol{a}) = \mathbb{E}_\pi[G_t \mid \boldsymbol{s}_t = \boldsymbol{s}, \boldsymbol{a}_t = \boldsymbol{a}]$ represents the expected discounted return when starting in state $\boldsymbol{s}$, taking action $\boldsymbol{a}$, and then following policy π. The state-value function $v_\pi(\boldsymbol{s}) = \mathbb{E}_\pi[G_t \mid \boldsymbol{s}_t = \boldsymbol{s}] = \max_{\boldsymbol{a}\in\mathcal{A}(\boldsymbol{s})} q_\pi(\boldsymbol{s}, \boldsymbol{a})$ represents the expected discounted return when starting in state $\boldsymbol{s}$ and following policy π. These value functions are generally not known, which makes the task of maximizing them very challenging. In order to find the optimal policy π^*, two general approaches can be taken: a value-based approach or a policy-based approach. We will describe both approaches briefly and introduce the two RL algorithms that are used in this thesis.

Value-based optimization using DQN A value-based approach to optimizing discounted returns consists of learning v_π or q_π and searching for a policy that maximizes these functions. An effective and popular algorithm that uses this approach is called Q-learning. It looks to find the policy $\pi^*(\boldsymbol{s}, \boldsymbol{a})$ that maximizes state-action values:

$$q^*(\boldsymbol{s}, \boldsymbol{a}) = \max_\pi \mathbb{E}_\pi[G_t \mid \boldsymbol{s}_t = \boldsymbol{s}, \boldsymbol{a}_t = \boldsymbol{a}] = \mathbb{E}_{\pi^*}[G_t \mid \boldsymbol{s}_t = \boldsymbol{s}, \boldsymbol{a}_t = \boldsymbol{a}] \tag{4.1}$$

As the Bellman equation holds for this optimal value function, this optimum can be found by iteratively applying the following update rule [Wat89], which is referred to as Q-learning:

$$q(\boldsymbol{s}_t, \boldsymbol{a}_t) \leftarrow q(\boldsymbol{s}_t, \boldsymbol{a}_t) + \alpha \left[r_{t+1} + \gamma \max_{\boldsymbol{a}} q(\boldsymbol{s}_{t+1}, \boldsymbol{a}) - q(\boldsymbol{s}_t, \boldsymbol{a}_t) \right] \tag{4.2}$$

Q-learning is a model-free off-policy algorithm with minimal convergence requirements[3] which can be met by applying an ϵ-greedy policy π_ϵ during training with a sufficient amount of iterations [SB18].

Deep Q-learning (DQN) is a value-based RL algorithm that applies value function approximation using deep neural networks. In order to deal with large state spaces and to enable generalization, DQN [MKS+13] applies a parametrized function (a neural network) to approximate the state-value function $Q_{\theta_{q,i}}(\boldsymbol{s}, \boldsymbol{a}) = Q(\boldsymbol{s}, \boldsymbol{a}; \theta_{q,i})$ at iteration i. The parameters θ_q of this 'Q-network' can be learned based on experience gained under an behavior policy π_ϵ by applying Stochastic Gradient Descent, using the loss function:

$$\mathcal{L}_i^{DQN}(\theta_{q,i}) = \mathbb{E}_{\pi_\epsilon}\left[\Big(\overbrace{\left[r_{t+1} + \gamma \max_{\boldsymbol{a}} Q\left(\boldsymbol{s}_{t+1}, \boldsymbol{a}; \theta^-_{q,i}\right) \right]}^{y_t} - Q(\boldsymbol{s}_t, \boldsymbol{a}_t; \theta_{q,i}) \Big)^2 \right] \tag{4.3}$$

Various techniques are used to avoid instability during training. A target network $Q(\boldsymbol{s}, \boldsymbol{a}; \theta^-_{q,i})$ is used to estimate the target values y_t during training. Its parameters θ^- are fixed during gradient steps, only to be updated with the policy network parameters θ every ζ^{DQN} steps. In order to de-correlate update sequences and address their non-stationary distribution, an experience replay mechanism can be used that uniformly samples transitions from the most recent episodes to generate training batches. These transitions are stored in a fixed-size replay buffer $\mathcal{M}^{DQN} = \{(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1}, r_{t+1}, d_{t+1})_{1..m}\}$, with size m and d indicating whether $\boldsymbol{s}_{t+1}$ is a terminal state. The core algorithm for Deep Q-Learning with experience replay is introduced in [MKS+13]. We specify the algorithm as it is applied in our specific setting in section 5.3.

Policy based optimization using PPO Unlike value-based methods that infer an optimal policy from a learned value function, Policy Gradient methods learn a differentiable parametrized policy function $\pi_{\theta_a}(\boldsymbol{a}_t|\boldsymbol{s}_t) = \pi(\boldsymbol{a}_t|\boldsymbol{s}_t; \theta_a)$ directly by optimizing the RL objective function:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[G_0(\tau) \mid \boldsymbol{s}_0] = V_{\pi_\theta}(\boldsymbol{s}_0) \tag{4.4}$$

[3] It requires that all possible state-action pairs have a non-zero probability of being updated

Typically, π_{θ_a} is referred to as the actor policy. We drop the subscript a for brevity whenever the type of parametrization is is clear from context. Methods that optimize $J(\theta)$ build on a general result of the Policy Gradient Theorem (proof in [SB18]) which states that the discounted policy gradient for episodic tasks is given by:

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T}\gamma^t G_t \nabla_\theta \log \pi_\theta\left(\boldsymbol{a}_t \mid \boldsymbol{s}_t\right) \mid \boldsymbol{s}_0\right] \\ &= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T}\gamma^t Q_{\pi_\theta}\left(\boldsymbol{s}_t, \boldsymbol{a}_t\right) \nabla_\theta \log \pi_\theta\left(\boldsymbol{a}_t \mid \boldsymbol{s}_t\right) \mid \boldsymbol{s}_0\right] \\ &\approx \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} Q_{\pi_\theta}\left(\boldsymbol{s}_t, \boldsymbol{a}_t\right) \nabla_\theta \log \pi_\theta\left(\boldsymbol{a}_t \mid \boldsymbol{s}_t\right)\right]\end{aligned} \tag{4.5}$$

This is a crucial result since it provides an expression for the gradient of the objective function, *independent* of the environment's dynamics P. The term $\log \pi_\theta\left(\boldsymbol{a}_t \mid \boldsymbol{s}_t\right)$ is referred to as the *score function*. Note that the term γ^t tends to be dropped in most popular RL algorithms which introduces bias in the gradient estimate, but it has been empirically shown to be effective in the episodic setting [NT20]. We also drop the conditional dependence on $\boldsymbol{s}_0$ in notation for convenience. The gradient estimator Eq.4.5 suffers from high variance. To remedy this, a baseline estimate $b(\boldsymbol{s}_t)$ can be subtracted from $Q_{\pi_\theta}(\boldsymbol{s}_t, \boldsymbol{a}_t)$ in this equation without introducing bias [Ben19], as long as $b(\boldsymbol{s}_t)$ is independent of $\boldsymbol{a}_t$. An intuitive choice for a baseline is the value estimate $b(\boldsymbol{s}_t) = V_{\pi_\theta}(\boldsymbol{s}_t)$ giving rise to the advantage function $A_{\pi_\theta} = Q_{\pi_\theta}(\boldsymbol{s}_t, \boldsymbol{a}_t) - V_{\pi_\theta}(\boldsymbol{s}_t)$. Advantage values can be interpreted as indicators of how much better or worse the observed result of an action during a roll-out was than expected. The policy gradient is scaled based on this result such that it shapes the policy function towards the best actions:

$$\nabla_\theta J\left(\theta\right) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} A_{\pi_\theta}\left(\boldsymbol{s}_t, \boldsymbol{a}_t\right) \nabla_\theta \log \pi_\theta\left(\boldsymbol{a}_t \mid \boldsymbol{s}_t\right)\right] \tag{4.6}$$

The advantage function is unknown when training an RL agent using this policy gradient method. In practice, it is often estimated using Generalized Advantage Estimation (GAE)[SMea16]:

$$\hat{A}_t^{GAE(\gamma,\lambda)} := \sum_{k=0}^{T-t}(\gamma\lambda)^k \delta_{t+k}^V = \sum_{k=0}^{T-t}(\gamma\lambda)^k \left(r_{t+k+1} + \gamma V_{\pi_\theta}\left(\boldsymbol{s}_{t+k+1}\right) - V_{\pi_\theta}\left(\boldsymbol{s}_{t+k}\right)\right) \tag{4.7}$$

Here, the parameter $\lambda \in [0, 1]$ can be chosen to trade-off bias and variance of the estimator. If $\lambda = 1$, Eq.4.7 reduces to $\hat{A}_t^{GAE(\gamma,1)} = G_t - V_{\pi_\theta}\left(\boldsymbol{s}_t\right)$ which is the same as 4.6 and has a low bias but a high variance. If $\lambda = 0$, we have $\hat{A}_t^{GAE(\gamma,0)} = r_{t+1} + \gamma V_{\pi_\theta}\left(\boldsymbol{s}_{t+1}\right) - V_{\pi_\theta}\left(\boldsymbol{s}_{t+1}\right) = \delta_t^V$ which is the TD-residual and has a high bias and low variance. The value function $V_{\pi_\theta}(\boldsymbol{s}_t)$ is unknown and needs to be estimated. This can be done by introducing a parametrized value function $\hat{V}_{\pi_\theta}\left(\boldsymbol{s}_t\right) = V_{\pi_{\theta_a}}(\boldsymbol{s}_t; \theta_c)$, which leads to the formulation of Actor-Critic methods. While operating under an Actor policy π_{θ_a}, parametrized by θ_a, a Critic (value function) is used, that depends on the actor policy, and is parametrized by θ_c. Both parameter sets θ_a and θ_c are leaned during training.

Proximal Policy Optimization (PPO)[SWD$^+$17] is a policy gradient-based RL algorithm that avoids sample inefficiency and training instability, which are two phenomena typically observed in vanilla policy gradient methods. The unconstrained gradient of the objective w.r.t. policy parameters θ was introduced in Eq.4.6. A gradient estimate $\hat{g} = \widehat{\nabla_\theta J(\theta)}$ can be calculated by taking the empirical average over a finite batch of samples obtained by interacting with the environment, which can be denoted as $\hat{g} = \hat{\mathbb{E}}_t\left[\hat{A}_t \nabla_\theta \log \pi_\theta\left(\boldsymbol{a}_t \mid \boldsymbol{s}_t\right)\right]$. Ideally, we would have a more data-efficient method that allows us to take multiple gradient steps using these computationally expensive sequences. This can be achieved by applying importance sampling, leading to:

$$\hat{g} = \nabla_\theta L_t^{CPI}(\theta), \text{ with } L_t^{CPI}(\theta) = \hat{\mathbb{E}}_t\left[\hat{A}_t \frac{\pi_\theta\left(\boldsymbol{a}_t \mid \boldsymbol{s}_t\right)}{\pi_{\theta_{\text{old}}}\left(\boldsymbol{a}_t \mid \boldsymbol{s}_t\right)}\right] = \hat{\mathbb{E}}_t\left[\hat{A}_t r_t(\theta)\right] \tag{4.8}$$

Here, L_t^{CPI} represents the surrogate ('conservative policy iteration') objective function and $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ denotes a probability ratio where θ_{old} represents the actor parameters before a policy update. We have obtained an unbiased[4] estimate of the policy gradient using off-policy samples from an older policy $\pi_{\theta_{\text{old}}}$. Large changes in policy parameters due to large gradient updates are avoided by constraining the size of a policy update using a clipping operation:

$$L_t^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(\hat{A}_t \cdot r_t(\theta), \hat{A}_t \cdot \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) \right) \right] \tag{4.9}$$

Clipping introduces a new hyperparameter ϵ, usually set between $[0.1, 0.2]$. The advantage estimation $\hat{A}_t$ can be based on GAE (Eq.4.7). Any differentiable architecture for the Actor and Critic models can be used, potentially sharing parameters between θ_a and θ_c. Gradient updates for actor and critic can be calculated simultaneously by maximizing a combined objective as a function of $\theta = \theta_a \cup \theta_c$:

$$L_t^{PPO}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S_t \left[\pi_\theta\right](s_t) \right] \tag{4.10}$$

with a loss term used for updating the critic:

$$L_t^{VF}(\theta) = \left(V_\theta(s_t) - V_t^{\text{targ}} \right)^2 \text{ with } V_t^{\text{targ}} = \sum_{k=0}^{T-t} \gamma^k r_{t+k+1} \tag{4.11}$$

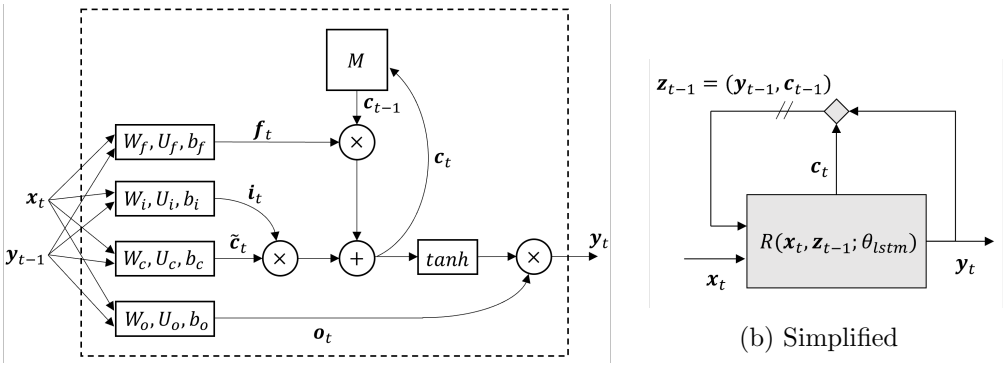
and an entropy term that can be used to reward higher-entropy action distributions and hence stimulate sufficient exploration:

$$S_t\left[\pi_\theta\right](s_t) = -\sum_{a \in \mathcal{A}} \pi_\theta(a \mid s_t) \log \pi_\theta(a \mid s_t) \tag{4.12}$$

Constants c_1, c_2 are hyperparameters used to mix the relative importance of the three objectives. In implementations of the PPO algorithm, the expectation $\hat{\mathbb{E}}_t$ of the losses are estimated by averaging losses obtained from minibatches of rollouts created under the policy π_θ. The core algorithm for PPO is introduced in [SWD+17]. We will specify the algorithm as it is applied in our specific setting in section 5.3.

Partial observability and memory-based state reconstruction Real-world control problems often involve noisy or sporadic (partial) observation of the environment's state. In a Partially Observable Markov Decision Process (POMDP), an agent does not have access to the true state of the environment, due to limitations in sensing capability. A POMDP [KLC98, HS15] can be described by $\mathcal{M}^{PO} = \langle \mathcal{S}, \mathcal{A}(\boldsymbol{s}), p_0, P, R, \gamma, \Omega, \mathcal{O} \rangle$ where the agent receives observations $\mathbf{o}_t \in \Omega$ according to some probability distribution $\mathbf{o}_t \sim \mathcal{O}(\boldsymbol{s}_t)$. Much study has gone into methods that aim to reconstruct the true state with some internal representation $\hat{\boldsymbol{s}}_t$ based on a history of observations $\boldsymbol{h}_t = (\boldsymbol{a}_0, \mathbf{o}_1, ..., \boldsymbol{a}_{t-1}, \mathbf{o}_t)$ acquired during an episode. We consider two methods that use apply such a form of feature approximation. A first method, frame stacking [MKS+13], introduces memory by stacking the k most recent observations (frames) and use this as an enhanced input to the model. This requires no change in model architecture, other than an increase in parameters associated with the first transformation of the input. Hence, it assumes the existing model is flexible enough implicitly learn state reconstruction based on the stacked input. A second method applies a Recurrent Neural Network, either to observations directly or to latent features (embedddings) anywhere in the network. We will use LSTM modules [HS97] given their advantageous properties and their demonstrated application in RL [HS15, WFPS07].

[4] assuming $\hat{A}_t$ is unbiased



(a) Full

(b) Simplified

Figure 4.2: Computational graph of an LSTM module (adapted from [BBCV21])

LSTMs can in theory learn to incorporate a full history and can be applied to signals anywhere in a dynamical system. An LSTM layer is characterized by a memory cell and internal signal modulation through the use of input, output and forget gates, see Fig.4.2. The definition of the update rule is as follows:

$$\begin{aligned}
\widetilde{\mathbf{c}}_t &= \tanh\left(\mathbf{W}_c \mathbf{x}_t + \mathbf{U}_c \mathbf{y}_{t-1} + \mathbf{b}_c\right) \\
\mathbf{i}_t &= \sigma\left(\mathbf{W}_{\mathrm{i}} \mathbf{x}_t + \mathbf{U}_{\mathrm{i}} \mathbf{y}_{t-1} + \mathbf{b}_{\mathrm{i}}\right) \\
\mathbf{f}_t &= \sigma\left(\mathbf{W}_{\mathrm{f}} \mathbf{x}_t + \mathbf{U}_{\mathrm{f}} \mathbf{y}_{t-1} + \mathbf{b}_{\mathrm{f}}\right) \\
\mathbf{o}_t &= \sigma\left(\mathbf{W}_{\mathrm{o}} \mathbf{x}_t + \mathbf{U}_{\mathrm{o}} \mathbf{y}_{t-1} + \mathbf{b}_{\mathrm{o}}\right) \\
\mathbf{c}_t &= \mathbf{i}_t \odot \widetilde{\mathbf{c}}_t + \mathbf{f}_t \odot \mathbf{c}_{t-1} \\
\mathbf{y}_t &= \mathbf{o}_t \odot \tanh\left(\mathbf{c}_t\right),
\end{aligned} \tag{4.13}$$

where $\mathbf{W}_\cdot, \mathbf{U}_\cdot, \mathbf{b}_\cdot$ are learnable weights and biases, $\sigma(x) = \frac{1}{1+e^{-x}}$ is the logistic sigmoid activation function, $\odot$ the element-wise multiplication. If we group all learnable weights into θ_{lstm}, this update rule can be summarized as:

$$\begin{aligned}
\mathbf{z}_t &= R(\mathbf{x}_t, \mathbf{z}_{t-1}; \theta_{lstm}), \\
\text{with } \mathbf{z}_t &= (\mathbf{y}_t, \mathbf{c}_t)
\end{aligned} \tag{4.14}$$

which is the simplified notation we will use when we apply LSTMs.

4.2 Graph representation learning

We follow the naming and notation conventions used in [BBCV21]. A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, consists of a set $\mathcal{V}$ of $n = |\mathcal{V}|$ nodes and a set $\mathcal{E} \subseteq \{(u, v) \mid u \in \mathcal{V}, v \in \mathcal{V}\}$ of $m = |\mathcal{E}|$ edges between nodes, where each edge $e = (u, v) \in \mathcal{E}$ is an unordered pair. A graph structure (its connectivity) can be described by the adjacency matrix $\boldsymbol{A} \in \{0, 1\}^{n \times n}$ with $a_{uv} = 1$ if $(u, v) \in \mathcal{E}$ and 0 otherwise. Equivalently, the edge index $\boldsymbol{E} \in \mathcal{V}^{2 \times m}$ forms a sparse description of this connectivity and holds all pairs $(u, v) \in \mathcal{E}$. The (directed) *1-hop* neighborhood of node u is the set with all accessible nodes from u, $\mathcal{N}_u = \{v : (u, v) \in \mathcal{E}\}$. The out-degree of a node u is defined as $d^{out}_{\mathcal{G}}(u) = d^{out}_u = |\mathcal{N}_u|$. Node characteristics are encoded by node features $\boldsymbol{x}_u \in \mathbb{R}^f, \forall u \in \mathcal{V}$, collectively forming the node feature matrix $\boldsymbol{X} = (\boldsymbol{x}_1, ...\boldsymbol{x}_n)^T \in \mathbb{R}^{n \times f}$ using an arbitrary (but fixed) ordering. The neighborhood features of node u are contained in a multi-set $\boldsymbol{X}_{\mathcal{N}_u} = \{\{\boldsymbol{x}_v : v \in \mathcal{N}_u\}\}$.

A Graph Neural Network (GNN) layer is defined as a function $F(\boldsymbol{X}, \mathbf{E}; \theta_{gnn})$ that applies a shared local function (a "diffusion", "propagation", or "message passing" operation) $\phi(\boldsymbol{x}_u, \boldsymbol{X}_{\mathcal{N}_u}; \theta_{gnn})$ to the features of all individual nodes and their respective neighbourhoods.

$$\boldsymbol{h}_u = \phi(\boldsymbol{x}_u, \boldsymbol{X}_{\mathcal{N}_u}; \theta_{gnn}) \tag{4.15}$$

The resulting latent node embeddings $\boldsymbol{h}_u \in \mathbb{R}^{d'}$ are stacked into the matrix $\boldsymbol{H} = F(\boldsymbol{X}, \mathbf{E}; \theta_{gnn}) = (\boldsymbol{h}_1, ..., \boldsymbol{h}_n)^T \in \mathbb{R}^{n \times d'}$. The edge index matrix $\mathbf{E}$ appears as an argument of F because it contains the structural information of the graph needed to determine node neighborhoods. The propagation function ϕ should be a permutation invariant function, meaning its outcome should not depend on the order in which the node features are presented in $\boldsymbol{X}_{\mathcal{N}_u}$. The resulting GNN layer F should be permutation equivariant, which is achieved by stacking the latents $\boldsymbol{h}_u$ into $\boldsymbol{H}$ using the same ordering as the node feature ordering in $\boldsymbol{X}$. Multiple GNN layers can be applied in succession to form a GNN, increasing the 'receptive field' of each node: nodes can receive incoming information that are at least k hops away, with k the number of layers of the GNN.

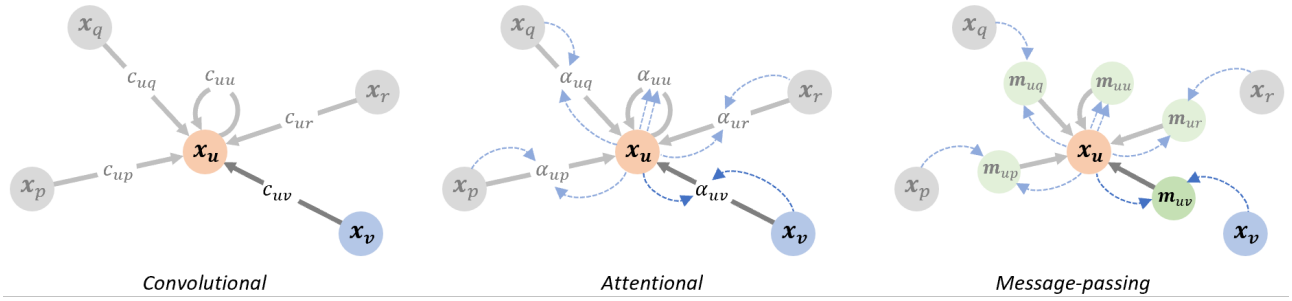


Figure 4.3: Generic propagation function formulations (adapted from [Vel21])

The expressiveness and usefulness of GNN layers are dependent on the choice of propagation function ϕ and the way in which the layers F are integrated into deep learning model architectures. [Vel21] distinguishes three categories of propagation functions with increasing levels of complexity, depicted in Fig.4.3. In the convolutional model $\boldsymbol{h}_u = \phi\left(\boldsymbol{x}_u, \bigoplus_{v \in \mathcal{N}_u} c_{uv} \psi\left(\boldsymbol{x}_v\right)\right)$, incoming information from neighboring nodes $\psi(\boldsymbol{x}_v)$ is aggregated using fixed weights c_{uv}. Usually, the aggregation function $\bigoplus_{v \in \mathcal{N}_u}$ is a summation and the weights depend on the graph structure (encoded by $\boldsymbol{E}$). The attentional model $\boldsymbol{h}_u = \phi\left(\boldsymbol{x}_u, \bigoplus_{v \in \mathcal{N}_u} a\left(\boldsymbol{x}_u, \boldsymbol{x}_v\right) \psi\left(\boldsymbol{x}_v\right)\right)$ adds flexibility by introducing a self-attention mechanism a that learns aggregation weights: $\alpha_{uv} = a(\boldsymbol{x}_u, \boldsymbol{x}_v)$. The message passing variant allows the calculation of messages $m_{uv} = \psi(\boldsymbol{x}_u, \boldsymbol{x}_v)$ by a learnable function ψ, again, aggregated to yield the embeddings: $\boldsymbol{h}_u = \phi\left(\boldsymbol{x}_u, \bigoplus_{v \in \mathcal{N}_u} \psi\left(\boldsymbol{x}_u, \boldsymbol{x}_v\right)\right)$.

Long distance information propagation by GNNs can be problematic. It has been shown [AY21] that stacking too many GNN layers in order to increase the receptive field can lead to information bottlenecks. This effect ('oversquashing') can hinder GNN performance. Hence, a key challenge in designing GNN-based architectures for combinatorial optimization problems on graphs is to ensure that such models are expressive enough to capture crucial structural patterns in the data and yet can be scaled to function on larger problem instances (i.e., larger graphs). We will experiment with two GNN formulations that have proven effective in various node selection tasks.

Structure2vec Originally developed to replace kernel methods for inference on structured data [DDS16], *struc2vec* showed promise in graph optimization tasks [KDZ+17] that require a set of nodes to be selected in order, such as Minimum Vertex Cover, Maximum Cut and the Travelling Salesman Problem. The method develops node embeddings recursively using an update function parametrized by θ with the following structure:

$$\boldsymbol{h}_u^{(\tau+1)} := \phi\left(\boldsymbol{x}_u, \left\{\boldsymbol{h}_v^{(\tau)}\right\}_{v \in \mathcal{N}_u}, \{w(u,v)\}_{v \in \mathcal{N}_u}; \theta\right) \tag{4.16}$$

Note that here, $\tau = 0, ..., \tau_{max} - 1$ denotes the iterative application of the function ϕ. The embeddings are initialized as zero vectors $\boldsymbol{h}_u^{(0)} = \mathbf{0}$ and the raw feature vector $\boldsymbol{x}_u$ is re-used at each iteration. For convenience, we have omitted a subscript t to indicate that these embeddings are calculated at each timestep during sequential decision processes when used in RL. Scalars $w(u, v)$ represent the edge

weights to node u's neighbors. Thoughout this work, we apply the following *struc2vec* parametrization:

$$\boldsymbol{h}_u^{(\tau+1)} = \text{ReLU}\left(\theta_1 \boldsymbol{x}_u + \theta_2 \sum_{v \in \mathcal{N}_u} \boldsymbol{h}_v^{(\tau)} + \theta_3 \sum_{v \in \mathcal{N}_u} \text{ReLU}\left(\theta_4 w(v,u)\right)\right) \tag{4.17}$$

with $\theta_1 \in \mathbb{R}^{d' \times f}$ mapping the node features $\boldsymbol{x}_u$ to dimension d' of the node embeddings' latent space, $\theta_2, \theta_3 \in \mathbb{R}^{d' \times d'}$, $\theta_4 \in \mathbb{R}^{d'}$. Naturally, all node features and node embeddings are dependent on time. During the processing one one particular timestep, the propagation function 4.17 is applied iteratively τ_{max} times.

Graph Attention Networks (GAT) GATs were introduced by [VCC+18] as a powerful generalization of transformers[5]. We will consider *GATv2* [BAY22], a dynamic graph attention variant that is strictly more expressive than the original formulation. *GATv2* follows the generic form introduced above, but now written as the τ-th layer of the GNN with $\boldsymbol{h}_i^{(\tau+1)} \in \mathbb{R}^{d'}$ and $\boldsymbol{h}_i^{(\tau)} \in \mathbb{R}^{d}$, $i \in \mathcal{V}$[6].

$$\boldsymbol{h}_u^{(\tau+1)} = \sigma\left(\sum_{v \in \mathcal{N}_u} \alpha_{uv} W \boldsymbol{h}_v^{(\tau)}\right), \tag{4.18}$$

where σ is the sigmoid function. In order to construct the attention weights α_{uv}, a scoring function e is defined for every incoming edge (v,u) that learns set a 'relevance' of the features of neighboring node v to node u for which the embedding is being calculated:

$$e(\boldsymbol{h}_u^{(\tau)}, \boldsymbol{h}_v^{(\tau)}) = \boldsymbol{a}^T \,\text{LeakyReLU}\left(\left[W||W\right] \cdot \left[\boldsymbol{h}_u^{(\tau)}||\boldsymbol{h}_v^{(\tau)}\right]\right), \tag{4.19}$$

where $\boldsymbol{a} \in \mathbb{R}^{d'}$ and $W \in \mathbb{R}^{d' \times d}$ are the learnable parameters, and $||$ is the concatenation operator. A softmax is used to normalize the scores into attention weights:

$$\alpha_{uv} = \text{Softmax}_v\left(e\left(\boldsymbol{h}_u^{(\tau)}, \boldsymbol{h}_v^{(\tau)}\right)\right) = \frac{\exp\left(e\left(\boldsymbol{h}_u^{(\tau)}, \boldsymbol{h}_v^{(\tau)}\right)\right)}{\sum_{v' \in \mathcal{N}_u} \exp\left(e\left(\boldsymbol{h}_u^{(\tau)}, \boldsymbol{h}_{v'}^{(\tau)}\right)\right)} \tag{4.20}$$

GATs can be expanded with *multihead attention* allowing multiple ways to 'weigh the neighborhood' when forming latent representations. This is achieved by using K independent attention mechanisms that each apply the transformation of Eq.4.18 and concatenate their output to produce $\boldsymbol{h}_u^{(\tau+1)} \in \mathbb{R}^{Kd'}$:

$$\boldsymbol{h}_u^{(\tau+1)} = \Big\Vert_{k=1}^{K} \sigma\left(\sum_{v \in \mathcal{N}_u} \alpha_{uv}^k \mathbf{W}^k \boldsymbol{h}_v^{(\tau)}\right) \tag{4.21}$$

In the final (prediction) layer of a GAT network, averaging is used in Eq.4.21 instead of concatenation (with the sigmoid function applied after averaging) to yield the embeddings $\boldsymbol{h}_u = \boldsymbol{h}_u^{(final)} \in \mathbb{R}^{d'}$.

[5] https://thegradient.pub/transformers-are-graph-neural-networks/

[6] The input of the first GNN layer are the raw node features, $\boldsymbol{h}_u^{(0)} = \boldsymbol{x}_u \in \mathbb{R}^{d=f}$

Chapter 5

Methods

Our research question leads us to investigate whether an appropriately modeled escape agent, trained using RL, is capable of exploiting weaknesses in the strategy of pursuit agents on a graph. The escape agent aims to reach certain target nodes with some level of urgency, while the pursuers aim to capture the escaper. The escaper may obtain sporadic information on the positions of some of the pursuers.

In order to answer the research question, we will conduct experiments that evaluate the performance of an escape agent, trained under different circumstances, whose behavior is either modeled directly by a parametrized policy $\pi_\theta(a|\boldsymbol{s})$ or deduced from a learned action-value function $\pi(\boldsymbol{s}) = \arg\min_{a'} q_\theta(\boldsymbol{s}, a')$. The aim is that such a policy π informs a best action (moving to a reachable node from the escaper's current position) for any graph state such that the escaper is able to reach its objective under a wide range of circumstances. In order to find successful behaviors, we will be looking to define policy models that can learn to represent the graph state in a high-dimensional latent space and transform this representation to action probabilities or state-action value estimates.

We train these models using two selected Reinforcement Learning algorithms: DQN and PPO. Training requires a simulation environment that enables rollouts on different graph classes and topologies, populated by a varying number of pursuit agents that behave according to an external optimizer (IMA), unknown to the escaper. This, in turn, requires us to define a partially observable Markov Decision Process (POMDP), where state transitions are a function of this external optimizer. The performance of these trained escape agents are evaluated and their generalization properties tested along the various dimensions mentioned in the research question. The approaches to modeling, optimization and evaluation are described next.

5.1 Modeling the environment

The problem definition needs to be formalized as an environment (a POMDP) that captures its essential aspects with the lowest level of complexity. This requires various design choices, some of which have been informed by the expertise of experienced officers and dispatchers at the Dutch Police.

State Space A road network is modeled as a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$[1], where nodes $u, v \in \mathcal{V}$ represent intersections and directed edges $(u, v) \in \mathcal{E}$ represent connecting roads. We assume all graphs are reflexive: $(u, u) \in \mathcal{E}, \forall u \in \mathcal{V}$ and undirected: $(u, v) \in \mathcal{E} \implies (v, u) \in \mathcal{E}$, meaning an agent can always remain on the same node during transitions and one-way streets do not exist. Although $\mathcal{G}$ lives in a 2-dimensional Euclidean space, we ignore node coordinates and represent $\mathcal{G}$ by its adjacency matrix $\boldsymbol{A} \in \{0, 1\}^{n \times n}$ (or edge index $\boldsymbol{E} \in \mathcal{V}^{2 \times m}$) and assume all edges have the same

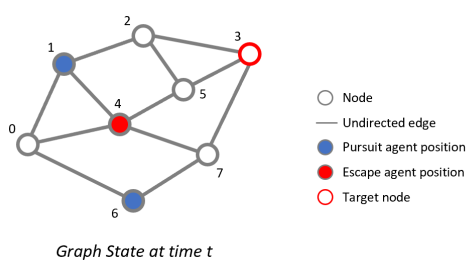


Figure 5.1: Environment definition

[1] Notation as introduced in section 4.2

edge weight, representing a unitary distance. We allow for the definition of node features $\boldsymbol{x}_u \in \mathbb{R}^f$ but do not consider additional edge features[2], although our methods allow for such an extension. These simplifications are justified by the fact that real-life road networks can be coarsened to a simplified abstract representation of interconnected and equidistant surveillance regions. The graph layout is fixed during a particular simulation instance but we may sample different graphs $\mathcal{G}$ from a graph set $\boldsymbol{\mathcal{G}}$ with categorical distribution $\mathcal{D}_{\mathcal{G}}$ when simulating episodes as part of an RL training regime. The graph $\mathcal{G}$ is populated by an escape agent whose node position at timestep t is given by $e_t \in \mathcal{V}$ and by U pursuit agents with node positions $\boldsymbol{u}_t = (u_t^{(1)}, ..., u_t^{(U)})$, $u_t^{(i)} \in \mathcal{V}$. The escaper's objective is to reach any of the target nodes $\mathcal{T} \subseteq \mathcal{V}$ ($\mathcal{T}$ is assumed stationary for a given graph instance). The state space $\mathcal{S} := \mathcal{V} \times \mathcal{V}^U \times \{0,1\}^{n \times n} \times 2^{\mathcal{V}}$ spans all agents' position permutations $\mathcal{V} \times \mathcal{V}^U$, given $\boldsymbol{A}$ and $\mathcal{T}$. The state $\boldsymbol{s} \in \mathcal{S}$ at timestep t on graph $\mathcal{G}$ is defined as:

$$\boldsymbol{s}_t = (e_t, \boldsymbol{u}_t, \boldsymbol{A}, \mathcal{T}) \tag{5.1}$$

Action Space We assume that only one single escape agent is present on the graph. The escaper's action space depends on its position e_t on $\mathcal{G}$ and consists of all reachable (neighboring) nodes. So we have: $a_t \in \mathcal{A}(\boldsymbol{s}_t)$ and $\mathcal{A}(\boldsymbol{s}_t) := \mathcal{N}_{e_t} = \{v : (e_t, v) \in \mathcal{E}\}$. We enforce all graphs to be reflexive, so $(u, u) \in \mathcal{E}$ and $u \in \mathcal{N}_u$.

Transition Dynamics During a rollout of a particular escape trajectory on a given graph instance, we assume the escape agent's movement over nodes to be deterministic: $e_{t+1} = a_t$. Moreover, in each timestep, the graph structure $\boldsymbol{A}$ and escaper's target node set $\mathcal{T}$, associated with the instance, remain constant. This yields: $\boldsymbol{s}_{t+1} = (a_t, \boldsymbol{u}_{t+1}, \boldsymbol{A}, \mathcal{T})$. Escape- and pursuit agents move synchronously. The transition dynamics $\mathscr{I}$ for the pursuit agents are dictated by the IMA optimizer and unknown to the escape agent:

$$\boldsymbol{u}_{t+1} = \mathscr{I}(t, \tilde{t}, e_{\tilde{t}}, \boldsymbol{u}_{\tilde{t}}, \boldsymbol{A}, \tilde{\mathcal{T}}), \tag{5.2}$$

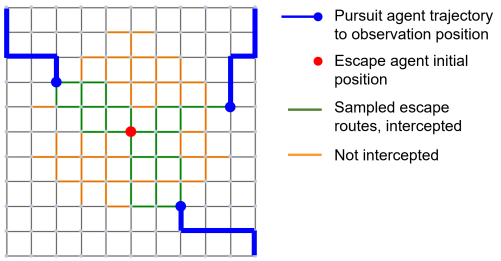


Figure 5.2: Baseline IMA generated pursuit trajectories (illustrative, $T^{\mathscr{I}}_{max} = 5$, $U = 3$)

where $\tilde{t}$ indicates the last timestep at which the position of the escaper was known to the pursuers and $\tilde{\mathcal{T}}$ is an assumption the pursuers may have on the objective of the escaper. There are various approaches that were developed for IMA and we will consider a baseline static search approach $\mathscr{I}_{base}$ with $\tilde{t} = 0$, $\tilde{\mathcal{T}} = \emptyset$ and an alternative approach $\mathscr{I}_{alt}$ with $\tilde{t} = 0$, $\tilde{\mathcal{T}} = \mathcal{T}$, presented in Appendix B. In short, the baseline approach calculates the optimal orchestration of pursuit units towards stationary observation positions, given the initial positions of all (escaper and pursuit) agents and a rollout time window of $T^{\mathscr{I}}_{max}$ steps. When $\tilde{\mathcal{T}} = \emptyset$, the pursuers do not assume any objective for the escaper (even though in reality, there may very well be a subset of nodes that the escaper is aiming to reach). In this case, pursuer routes are calculated by assuming the escaper performs a 'non-backtracking random walk'[3] (i.e., is not lingering). An example of such an orchestration is given in Fig.5.2 (examples for more complex graphs are shown in Fig.B.1-B.3 in the Appendix).

Terminal conditions We define three terminal conditions in our simulated environment:

1. After taking a step, the escape agent reaches a target node: $e_{t+1} \in \mathcal{T}$.

2. The escape agent is captured: either the escaper shares a node with at least one of the pursuers (a 'node capture'): $e_{t+1} \in \boldsymbol{u}_{t+1}$ or an escaper and a pursuer cross each other on an edge (an 'edge capture'): $\exists\, i$ s.t. $(e_t, e_{t+1}) = (u_{t+1}^{(i)}, u_t^{(i)})$.

[2] such as distance, speed limit, presence of an ANPR camera, etc.

[3] A non-backtracking random walk defined as a random walk in which a step is not allowed to go back to the immediate previous state: $e_{t+1} \neq e_t \implies e_{t+2} \neq e_t$ holds for any segment of the walk $(... \to e_t \to e_{t+1} \to e_{t+2} \to ...)$

3. Simulation time limit is reached: $t >= T_{max}$. This time limit needs to be set sufficiently large to enable an escape agent to adapt its course, stay strategically put or even backtrack on its way to a target node if this is beneficial. In practice we set T_{max} to twice the maximum distance[4] of all node pairs that have valid paths between them on $\mathcal{G}$.

In summary, the following terminal condition is used in the RL algorithms:

$$\begin{aligned} \text{done}_{t+1} = & \text{'captured'}: \Big[e_{t+1} \in \boldsymbol{u}_{t+1} \vee \exists\, i \text{ s.t. } (e_t, e_{t+1}) = (u_{t+1}^{(i)}, u_t^{(i)}) \Big] \vee \\ & \text{'escaped'}: \Big[e_{t+1} \in \mathcal{T} \Big] \vee \\ & \text{'out of time'}: \Big[t \geq T_{max} \Big] \end{aligned} \tag{5.3}$$

Reward Function We assume the escape agent combines multiple objectives. The primary goal of the escaper is to not get caught. The secondary goal is to reach any target node $u \in \mathcal{T}$ with some level of urgency. We encode these objectives by assigning a reward of $r_{capture} = -10$ when the escaper is captured, $r_{success} = +10$ when a target node is reached and $r_{step} = -0.1$ for each action the escaper takes[5]. The small step-based penalty introduces an incentive to reach a target node as soon as possible. Care should be to taken when setting this step-based penalty relative to the other rewards. If the distance d to a feasible target node is very long, it might become more profitable to incur an immediate capture penalty instead of accumulating future (discounted) step penalties: $\gamma^0 \cdot r_{capture} > \sum_{k=t}^{t+d} \gamma^{k-t} \cdot r_{step}$. We have ensured that the basic reward function described above does not lead to such incentives in any of the scenarios we will be considering.

Observation space and the Markov property In order to combine reinforcement learning with graph representation learning, we have to represent the state of the environment in a way that it can be processed by a GNN. For this purpose, we define an observation to be a tuple that consists of a node feature matrix and an edge index:

$$\boldsymbol{o}_t = (\boldsymbol{X}_t, \boldsymbol{E}), \tag{5.4}$$

with $\boldsymbol{X}_t = (\boldsymbol{x}_1, ...\boldsymbol{x}_n)_t^T \in \mathbb{R}^{n \times f}$ (see Sec.4.2). Note that this representation may encode more information than that contained in the state definition. The most basic node feature definition that captures all information contained in the full state $\boldsymbol{s}_t$ has:

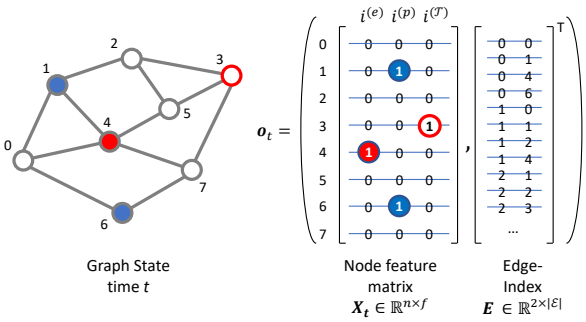


Figure 5.3: Basic node feature definition

$$\boldsymbol{x}_{v,t} = \left(i_{v,t}^{(e)}, i_{v,t}^{(p)}, i_{v,t}^{(\mathcal{T})}\right)^T \in \mathbb{R}^{f=3} \tag{5.5}$$

It assigns, for each timestep and each node $v \in \mathcal{V}$, integer values to represent: $i_{v,t}^{(e)} \in \{0, 1\}$ indicating whether the escape agent occupies node v; $i_{v,t}^{(p)} \in \{0, ..., U\}$ the number of pursuers that occupy node v and $i_{v,t}^{(\mathcal{T})} \in \{0, 1\}$ indicating whether node v is a target node. An example is given in Fig.5.3. Higher dimensional node features will be used later on, their definitions are given in Appendix D. Given the definitions above, we can reason about the conditions under which this model is a true Markov Decision Process. Whether the states and observations have the Markov property, depends on the nature of the transition dynamics $\mathscr{I}$ (Eq.5.2) of the pursuers. If an action a_t by the escaper, given a certain graph state $\boldsymbol{s}_t$ (a 'constellation' on the graph) always results in the same response probability for $\boldsymbol{s}_{t+1}$ and r_{t+1}, the Markov property holds. Should $\mathscr{I}$ be such that pursuers adapt their paths during rollouts based on new information, or execute certain repetitive patterns, the Markov property

[4]The distance between two vertices in a graph is the length of a shortest path between them (infinity if no path exists)

[5]Optionally, a $r_{step} = -0.15$ reward could be assigned if the action taken is to stay put on the same node, $a_t = e_t$. This addition would mimic the an assumed preference to be on the move

may no longer hold. In that case, stability and convergence properties of the RL algorithms may be affected. In our experiments, we will use IMA models that engage in static search campaigns in which case the Markov property holds. Based on the initial position of the escaper e_0, pursuers are directed to fixed observation positions using directed paths[6], irrespective of the escaper's actions.

Modeling Partial Observability In this work, we take the perspective of the escape agent. It is useful to consider to what extent an escaper has knowledge of the environment state $\boldsymbol{s}_t$ (Eq.5.1) since any form of optimization of behaviour will heavily depend on this. We assume the escaper is sophisticated and well prepared, with full knowledge of the road network (encoded in $\boldsymbol{A}$) and some pre-planned escape plan ($\mathcal{T}$) and obviously aware of its own position (e_t). However, it is unrealistic to assume that an escaper has full access to information on its pursuers[7]. Based on discussions with experienced police officers and dispatchers, we conclude that in a worst-case scenario, a sophisticated escaper may have real-time access to pursuer positions ($\boldsymbol{u}_t$) with some frequency, for instance by deploying spotters that gather such information, or by acquiring illegal access to C2000 radio transmissions. It would be difficult, however, for a human escaper to construct a full picture of all pursuer positions at each point in time in a real-world scenario. Therefore, we define the *escaper visibility of its pursuers* as a fixed probability p_{vis}. At each timestep, the position of each individual pursuit agent is observed with probability p_{vis} and we can use a masking operator to construct the second element $i_{v,t}^{(p)}$ of node features $\boldsymbol{x}_{v,t} = (i_{v,t}^{(e)}, i_{v,t}^{(p)}, i_{v,t}^{(\mathcal{T})}), \forall v \in \mathcal{V}$ under partial observability:

$$\begin{aligned} \text{given pursuer node positions: } & \boldsymbol{u}_t = (u_t^{(1)}, ..., u_t^{(U)}), \\ \text{set visibility mask for } j^{th} \text{ pursuer: } & m_t^{(j)} = \begin{cases} 1, \text{ with prob. } p_{vis} \\ 0, \text{ otherwise} \end{cases} \\ \text{calculate number of observed pursuers on node } v\text{: } & i_{v,t}^{(p)} = \sum_{j=1}^{U} m_t^{(j)} \cdot \mathbb{1}_{u_t^{(j)}=v} \end{aligned} \tag{5.6}$$

Examples that illustrate this method are presented in Appendix D. Note that under partial observation, the Markov property no longer holds (the same observations may reflect very different true states) and we will have to apply memory-based feature approximation (see section 4.1) in order to solve such a Partially Observable Markov Decision Process (POMDP).

5.2 Datasets

Our experiments require agent models to be trained using a dataset that contains a large number of graph instances from certain graph classes. Subsequently, we want to be able to evaluate those trained models on different graph instances from new and unseen graph classes. Such a dataset was not available and it had to be created as part of this project. There were several stages involved in creating the dataset. First, a data-structure was created that could efficiently contain all graph data, as well as pre-calculated pursuer responses for a range of initial conditions (the initial positions of all agents) of the graph instances. Pre-calculating pursuer responses was necessary to enable fast enough training. It was infeasible to use the existing IMA optimizer, which calls the Gurobi Mixed Integer Programming solver, at runtime. Depending on graph size, graph complexity and number of agents, calculating pursuer responses can take in the order of hundreds of seconds. Lastly, a seamless interface with the similation environment had to be built such that the generic function that performs a timestep in the simulation can instantaneously retrieve next state and reward values. More details on the implementation are given below.

Graph classes Fig.5.4 shows the four graph classes that were created as part of the datatset. The classes range in size, structure and complexity. Large scale city graphs were extracted from the

[6] A directed path is a walk on a graph following directed edges in which all edges and nodes are unique

[7] Such information may include position, direction and speed of travel, pursuer coordination strategy, etc.

Dutch road network registry[8] and post-processed for this application. An overview of graph statistical properties of these data can be found in Appendix C.

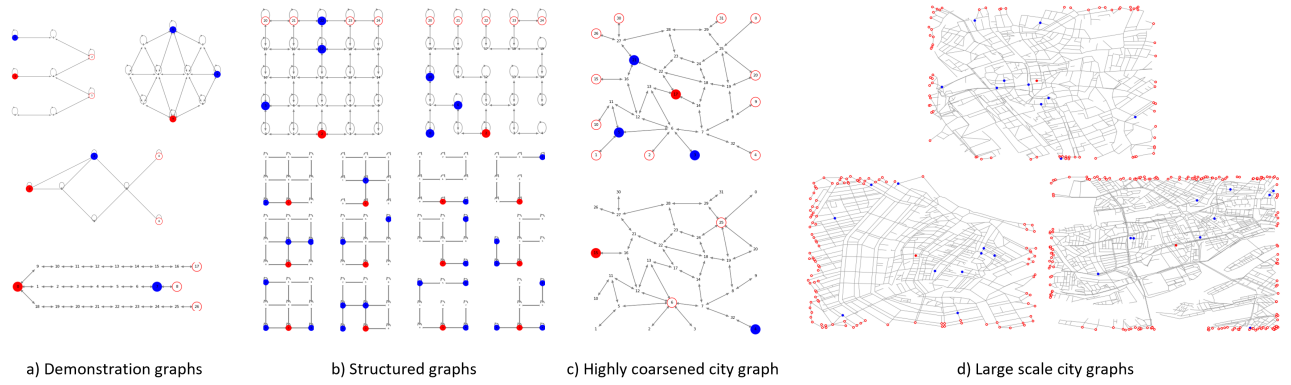


Figure 5.4: Overview of graph classes (a-d) used for our experiments, with illustrative instantiations of different topologies within these classes

A graph class may contain different graph topologies, and each topology can be instantiated with an initial state in multiple ways by varying i) the initial position of the escape agent, ii) the number of pursuit agents and their initial positions, iii) the number of target nodes and their locations. The resulting number of unique instances grows exponentially with graph size. However, we do not need to create exhaustive datasets that contain all possible instances; we can generate sufficiently large train-, validation- and testsets and sample from these. Note that we do not use spatial information in our node features, which means our GNN architecture is invariant to rotations, symmetries and isomorphisms. However, we keep these potential duplications in our graph dataset because the pursuer strategy $\mathscr{I}$ may in fact use spatial information. Pursuer responses $\boldsymbol{u}_{t+1}$ (Eq.5.2) will vary with each graph instance as they depend on the initial conditions. We assume pursuers only base their responses on the initial position of the escaper which means that all responses can be pre-calculated, which speeds up RL training. Examples of pursuer behavior are provided in Appendix B.

In order to accommodate detailed generalizability experiments on small graphs, a graph instance sampling mechanism was developed following the principles described in [HL13], using a full 3x3 Manhattan base graph. Segments of 8×3 graph topologies are defined based on i) permutations of the base graph by removing between $0-7$ edges, and ii) varying the number of pursuers on the permuted graph between $1-3$. As before, the initial positions of the escape agent and pursuers and the target node set can be assigned randomly to create specific graph instances. Pursuer responses can be pre-calculated for these instances. Fig.5.4(b, bottom) shows several samples from this mechanism, detailed graph class statistics are provided in Fig.C.5. Other graph classes besides the highly structured Manhattan graphs are shown in Fig.5.4(a-d). The full dataset will be made public, subject to approval by the Dutch Police, for future reference and experimentation.

Simulation environment A custom Reinforcement Learning environment `GraphWorld` was created to conduct our experiments, following the standards of OpenAI's `gym`[9] framework. This derived environment class implements all functions described in section 5.1. Observation wrappers are used to enable the flexible definition of node features. `GraphWold` instances contain one specific graph topology and load pursuer rollouts for all pre-calculated initial conditions. In addition, a `SuperEnv` class was created in which an arbitrary number of `GraphWorld` classes can be stacked such that graph instances and initial conditions can be randomly sampled for training purposes. This means an escape agent can be exposed to a wide range of experiences in terms of graph sizes, topologies and initial conditions during training. With the datasets and environment class in place, we can formulate the initiation of a simulation session. This is described in Algorithm 1 below.

[8]Dataset Nationaal Wegenbestand https://www.pdok.nl/
[9]https://gym.openai.com/

Algorithm 1 Subroutine: reset Search & Pursuit Evasion (SPE) environment

1: Draw graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}) \sim \mathcal{D}_{\mathcal{G}_{train/validation/test}}$, receive edge index $\boldsymbol{E}$
2: Draw random initial node positions of escape agent e_0 and pursuers $\boldsymbol{u}_0 = \{u_0^{(0)}, ..., u_0^{(U)}\}$ on $\mathcal{G}$, s.t. $e_0 \notin \boldsymbol{u_0}$
3: Draw subset of $n_{\mathcal{T}}$ target nodes $\mathcal{T} \subseteq \mathcal{V}$
4: Calculate node feature matrix $\boldsymbol{X}_0$ using Eq.D.1-D.8
5: Return observation $\boldsymbol{o}_0 = (\boldsymbol{X}_0, \boldsymbol{E})$, escaper initial position e_t

5.3 Integrated GNN-RL formulation

Now that a formal POMDP has been defined and datasets have been created, we move to the actual formulation of the escape agent model. This section will describe how an input (an observed graph state) can be transformed to produce escaper actions. We cast our Search & Pursuit-Evasion problem definition into an RL framework, where at any given state, a best action consists of selecting a reachable node on the graph, based on some transformation of a latent representation of the state of the graph. GNNs are used to learn such graph representations in the form of node- and graph embeddings. In the broad field of graph representation learning, node embeddings produced by GNNs can be applied to a variety of settings, involving such tasks as node/graph classification, node clustering or edge prediction. In our setup, the objective of selecting the next best action node selection can be interpreted as an edge prediction task.

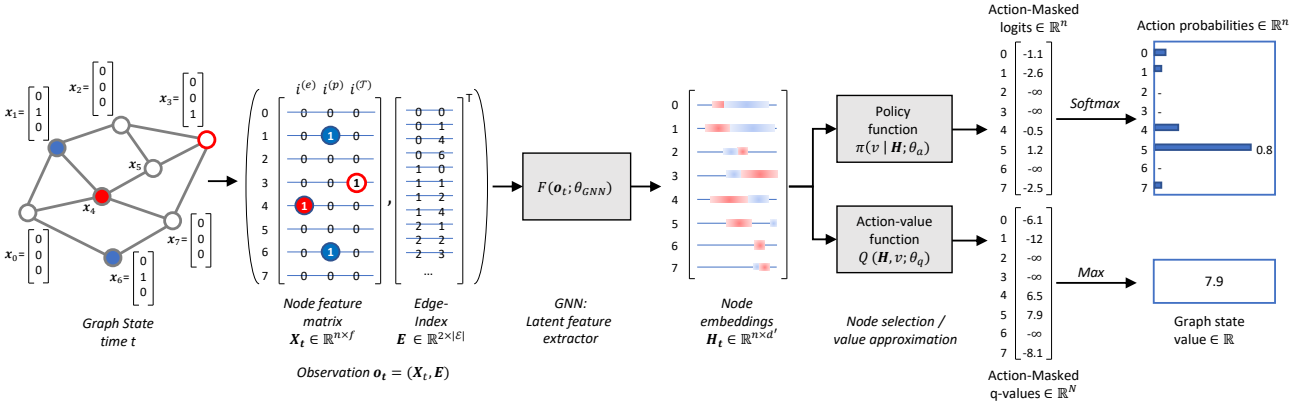


Figure 5.5: High-level overview escape route generation using a GNN-RL formulation

Consider an escape agent (indicated by the solid red node 4 in Fig.5.5) whose aim is to reach target node 3, with pursuer agents on nodes 1 and 6 (the blue nodes). A rudimentary way to encode this state into node features was defined in Eq.5.5 as $\boldsymbol{x}_{v,t} = (i_{v,t}^{(e)}, i_{v,t}^{(p)}, i_{v,t}^{(\mathcal{T})})^T$ which assigns escape- and pursuer agent presence $i_v^{(e)}, i_v^{(p)}$ and target node status $i_v^{(\mathcal{T})}$ to each node. Passing the input $(\boldsymbol{X}_t, \boldsymbol{E})$ through a GNN will yield node embeddings:

$$\boldsymbol{H}_t = F(\boldsymbol{X}_t, \mathbf{E}; \theta_{gnn}) = F(\boldsymbol{o}_t; \theta_{gnn}) \tag{5.7}$$

from which a latent representation of the graph state can be constructed. Both a policy function approximator $\pi(v \mid \boldsymbol{H}; \theta_a)$ and value function approximators $Q(\boldsymbol{H}, v; \theta_q)$, $V(\boldsymbol{H}; \theta_c)$ can be defined that produce action probabilities or q-values for all nodes $v \in \mathcal{V}$ of the graph, or a value for the state of the graph, respectively. This pipeline (Fig.5.5) can be thought of as three high-level stages: i) read the graph state, ii) interpret this state in a latent space using a GNN, iii) learn to predict a best action and state value based on this interpretation. This principle was successfully demonstrated for other combinatorial optimization and routing problems on graphs, see section 3.3. In those cases, a set ('partial solution') of previously selected nodes is used as part of the state encoding. In our SPE problem, we need to adjust this notion as it might be necessary to select nodes/edges multiple times. We also want a policy to be able to scale to arbitrary structured and sized graphs. This requires several adjustments to methods, notably the application of invalid action masking [HO20] and using mean pooling of node features as an aggregation method instead of summing used in [KDZ+17].

Function approximations The following parametrized functions are used in our experiments:

- Q-value approximation (Q-value for node v, given escape agent's node position $e_t = u$)

$$Q(\boldsymbol{H}, v; \theta_q) = \begin{cases} \theta_{q_1} \text{ ReLU}\left(\left[\theta_{q_2} \quad \overbrace{\frac{1}{n}\sum_{w\in\mathcal{V}} \boldsymbol{h}_w}^{\text{Graph representation}} \quad \middle\| \quad \theta_{q_3} \quad \overbrace{\boldsymbol{h}_v}^{\text{Node representation}} \quad \right]\right) & \text{, if } v \in \mathcal{N}_u \\ -\infty & \text{, otherwise} \end{cases} \tag{5.8}$$

with $\theta_q = \{\theta_{q1}, \theta_{q2}, \theta_{q3}\}$, and $\theta_{q_1} \in \mathbb{R}^{2d'}$ and $\theta_{q_2}, \theta_{q_3} \in \mathbb{R}^{d'\times d'}$. Note that the concatenation (the symbol $\|$ inside the brackets) in Eq.5.8 combines a transformation θ_{q2} on the 'graph representation' $\bar{\boldsymbol{h}} = \frac{1}{n}\sum_{w\in\mathcal{V}} \boldsymbol{h}_w$ (obtained by mean-pooling all node embeddings) and a transformation θ_{q3} on the target node representation $\boldsymbol{h}_v$ (node v's embedding). This combined representation can be calculated on an arbitrary sized graph using one set of weights, a crucial property that enables generalization across different graph topologies.

For Actor-Critic methods:

- An Actor model can be based on the same principle of transforming the node embeddings, but with different parameters:

$$\log \pi(v \mid \boldsymbol{H}; \theta_a) \text{ defined according to Eq.5.8, but with } \theta_a = \{\theta_{a_1}, \theta_{a_2}, \theta_{a_3}\} \tag{5.9}$$

- A Critic model can be obtained either by applying a linear transformation to the embeddings:

$$V^{lin}(\boldsymbol{H}; \theta_c) = \theta_c \frac{1}{n}\sum_{w\in\mathcal{V}} \boldsymbol{h}_w \tag{5.10}$$

with $\theta_c \in \mathbb{R}^{d'}$or by applying a maximum operator to Eq.5.8:

$$V^{max}(\boldsymbol{H}; \theta_c) = \max_{v\in\mathcal{V}} Q(\boldsymbol{H}, v; \theta_c) \tag{5.11}$$

We will experiment with different formulations with the aim of keeping the critic model as simple as possible, but expressive enough to ensure that it is a compatible function, a necessary condition for stability and convergence of the RL algorithms that we will be applying.

Algorithm 2 Deep Q-Learning with GNN based Search & Pursuit-Evasion (SPE)

Initialize:
  replay memory buffer $\mathcal{M}^{dqn}$ with capacity for m transitions
  weights $\theta = \{\theta_{gnn}, \theta_q\}$, $\theta^- = \{\theta^-_{gnn}, \theta^-_q\}$ of Q-network $Q(F(\boldsymbol{o}; \theta_{gnn}), v; \theta_q)$ and target network $Q(F(\boldsymbol{o}; \theta^-_{gnn}), v; \theta^-_q)$, Eq.5.8
  exploration Schedule $\mathcal{S}_\epsilon$, θ^- update frequency ζ^{dqn}
for episode κ=1...κ_{max} **do**
    Set $t = 0$, $done_0 = False$, $\epsilon \leftarrow \mathcal{S}_\epsilon(e)$, reset SPE environment (subroutine 1), receive $\boldsymbol{o}_t = (\boldsymbol{X}_0, \boldsymbol{E})$, e_t
    while not done **do**
        Take action a_t=$\begin{cases} \text{random node } v \in \mathcal{N}_{e_t}, \text{ with prob. } \epsilon \\ \text{argmax}_{v\in\mathcal{V}}\, Q(F(\boldsymbol{o}_t; \theta_{gnn}), v; \theta_q), \text{ Eq.5.8, otherwise} \end{cases}$
        Receive r_{t+1}, $e_{t+1} = a_t$, $\boldsymbol{u}_{t+1}$ from $\mathscr{I}$ (Eq.5.2), $\boldsymbol{o}_{t+1} = (\boldsymbol{X}_{t+1}, \boldsymbol{E})$, done$_{t+1}$ from Eq.5.3
        Store transition $(\boldsymbol{o}_t, a_t, r_{t+1}, \boldsymbol{o}_{t+1}, done_{t+1})$ in $\mathcal{M}^{dqn}$
        if size $\mathcal{M}^{dqn} \geq$ batch_size **then**
            Sample random minibatch of transitions $(\boldsymbol{o}_j, a_j, r_{j+1}, \boldsymbol{o}_{j+1}, done_{j+1})$ from $\mathcal{M}^{dqn}$
            Use TD(0) to set $y_j = \begin{cases} r_{j+1}, \text{ if } done_{j+1} \\ r_{j+1} + \gamma \max_{v\in\mathcal{V}} Q(F(\boldsymbol{o}_{j+1}; \theta^-_{gnn}), v; \theta^-_q), \text{ Eq.5.8, otherwise} \end{cases}$
            Perform SGD step to update θ using $\mathcal{L}^{DQN} = (y_j - Q(F(\boldsymbol{o}_j; \theta_{gnn}), a_j; \theta_q))^2$, Eq.4.3
            Every ζ^{dqn} SGD steps: set $\theta^- = \theta$
        end if
        t += 1
    end while
    With validation frequency τ^{val}, save checkpoint of best model during training
end for

Deep Q-Learning implementation (DQN) With the GNN-based models and the integrated GNN-RL framework defined above, we can now apply our selected RL algorithms. The DQN implementation under full observability follows Algorithm 2. We use both the *struc2vec* (Eq.4.17) and *GATv2* (Eq.4.21) GNN models to extract the latent features (node embeddings) based on the input, $\boldsymbol{H}_t = F(\boldsymbol{o}_t; \theta_{gnn})$. Note that F can act on any sized graph using the same set of model weights. This means that we can sample from any collection of graph classes that we decide to include in the training set $\boldsymbol{\mathcal{G}}_{train}$ under $\mathcal{D}_{\mathcal{G}_{train}}$, as long as we include the observations $\boldsymbol{o}_t = (\boldsymbol{X}_t, \boldsymbol{E})$ in the transitions that are stored in the replay memory buffer. We can further diversify experience gained during training by randomizing the initial conditions on the graphs (starting positions of the agents and target node set). Applying DQN successfully in our setting requires a careful choice of hyperparameters, in particular model size and architecture, replay memory size, update frequency and the epsilon-greedy exploration schedule. An overview of parameter values used in this study is provided in appendix D.

Learning under Partial Observability In any real-world scenario, an escape agent will suffer from partial observability of the state of the graph (specifically: the position of its pursuers). We formalized this notion by introducing a visibility probability p_{vis} and a masking operation that yields node features $\boldsymbol{x}_{v,t} = (i^{(e)}_{v,t}, i^{(u)}_{v,t}, i^{(\mathcal{T})}_{v,t})^T$ with some fraction of the pursuer position information contained in $i^{(u)}_{v,t}$ removed (Eq.5.6). Reconstruction of the full state using feature approximation requires the ability to tap into (part of) the history of observations $\boldsymbol{o}_{:t}$. This can be achieved by introducing some appropriate notion of memory. We consider two approaches:

- Frame-stacking simply extends the node features with selected information ('frames') from previous timesteps. In our case, we could remember the sightings of pursuer units on nodes $v \in \mathcal{V}$ going k timesteps back, and construct the extended node features:

$$\begin{aligned} \boldsymbol{x}_{v,t} &= (i^{(e)}_{v,t}, i^{(p)}_{v,t-k}, ..., i^{(p)}_{v,t-1}, i^{(p)}_{v,t}, i^{(\mathcal{T})}_{v,t})^T \in \mathbb{R}^{f=k+3} \\ \boldsymbol{X}_t &= (\boldsymbol{x}_1, ..., \boldsymbol{x}_n)^T_t \in \mathbb{R}^{n \times f} \end{aligned} \tag{5.12}$$

  One can now use the existing architecture but feed it with the extended node features. This increases the input dimensions and thus the number of trainable parameters. This method is simple and can be effective. However, the memory time span in inherently limited to the number of frames k. Details and examples of node feature definitions can be found in Appendix D.

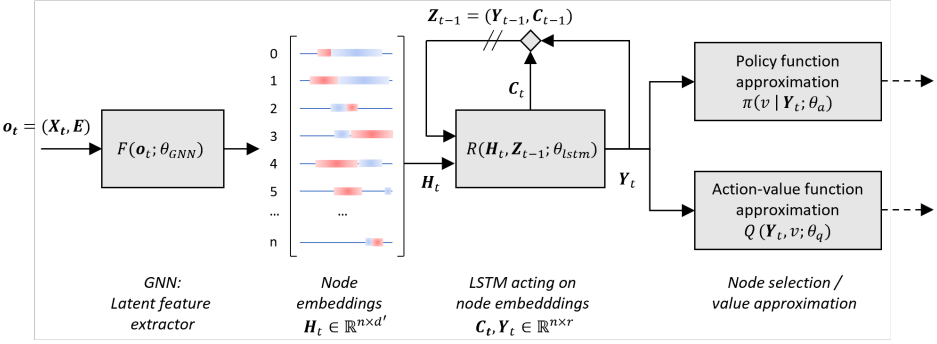


Figure 5.6: GNN-LSTM integration

- Alternatively, one can endow the escape agent with a memory module in the form of a recurrent neural network. We will use an LSTM module given its advantageous properties. LSTMs avoid vanishing or exploding gradients during back-propagation through time and are able to learn the optimal amount of time that past observations need to be remembered. We experiment with different positions of the LSTM. For now, we assume an LSTM with hidden dimension r is applied to the node embeddings and shared between Actor and Critic models as indicated in Fig.5.6. So we have (applying 4.14):

$$\boldsymbol{Z}_t = (\boldsymbol{Y}_t, \boldsymbol{C}_t) = R(\boldsymbol{H}_t, \boldsymbol{Z}_{t-1}; \theta_{lstm}) \tag{5.13}$$

We can rewrite the function approximations (Eq.5.8-5.11) as follows:

$$\pi(v \mid \boldsymbol{Y}_t; \theta_a) = \pi\left(v \mid R\left(\boldsymbol{H}_t, \boldsymbol{Z}_{t-1}\right); \theta_{lstm}, \theta_a\right) = \pi(v \mid \boldsymbol{o}_{:t}; \theta_{gnn}, \theta_{lstm}, \theta_a) = \pi_\theta(v \mid \tilde{\boldsymbol{o}}_{:t}) \tag{5.14}$$

$$Q(\boldsymbol{Y}_t, v; \theta_q) = Q(R\left(\boldsymbol{H}_t, \boldsymbol{Z}_{t-1}\right), v; \theta_{lstm}, \theta_q) = Q(\boldsymbol{o}_{:t}, v; \theta_{gnn}, \theta_{lstm}, \theta_q) = Q_\theta(\tilde{\boldsymbol{o}}_{:t}, v), \tag{5.15}$$

$$V(\boldsymbol{Y}_t; \theta_c) = V(R(\boldsymbol{H}_t, \boldsymbol{Z}_{t-1}); \theta_{lstm}, \theta_c) = V(\boldsymbol{o}_{:t}; \theta_{gnn}, \theta_{lstm}, \theta_c) = V_\theta(\tilde{\boldsymbol{o}}_{:t}) \tag{5.16}$$

with $\theta = \{\theta_{gnn}, \theta_{lstm}, \theta_a, \theta_c\}$ the full set of trainable weights of the model. The same parametrized functions Eq.5.8-5.11 can be used, but now acting on the 'memory reconstructed node embeddings' $\boldsymbol{Y}_t$ instead of the embeddings at current time $\boldsymbol{H}_t$, see Fig.5.6.

Algorithm 3 PPO with a GNN+LSTM based Search & Pursuit-Evasion agent

Initialize:
  rollout memory $\mathcal{M}^{ppo}$ with capacity $n_{rollout}$, transition memory $\mathcal{N}^{ppo}$ with capacity T_{max} transitions
  weights $\theta = \{\theta_{gnn}, \theta_{lstm}, \theta_a, \theta_c\}$ of feature extractor $F(\boldsymbol{o}_t; \theta_{gnn})$, Actor (Eq.5.14) and Critic (Eq.5.16)
for iteration it=1...n_{iter} **do**
  Generate rollouts:
  Clear $\mathcal{M}^{ppo}$
  for rollout r=1...$n_{rollout}$ **do**
    Set $t = 0$, $done_t = False$, reset environment (subroutine 1), receive $\boldsymbol{o}_t = (\boldsymbol{X}_0, \boldsymbol{E})$
    Clear $\mathcal{N}^{ppo}$, reset hidden cell state $\boldsymbol{Z}_{-1}^{(out)} = (\boldsymbol{0}, \boldsymbol{0})$.
    while not done **do**
      Generate transition $\xi_t = (\boldsymbol{o}_t, a_t, r_{t+1}, \boldsymbol{o}_{t+1}, done_{t+1}, \boldsymbol{Z}_t^{(in)}, \boldsymbol{Z}_t^{(out)})$ (subroutine 4) and store in $\mathcal{N}^{ppo}$
      t += 1
    end while
    store trajectory $\tau_r = (\xi_0, ..., \xi_{t-1})$ in $\mathcal{M}^{ppo}$, with transitions ξ_i taken from $\mathcal{N}^{ppo}$
  end for
  Perform gradient updates:
  Set $\theta^{old} = \theta$, $\mathcal{L}^{ppo} = 0$
  for each update step k=1...$n_{updates}$ **do**
    for each trajectory $\tau_i \in \mathcal{M}^{ppo}$ **do**
      for each transition $\xi_j \in \tau_i$ **do**
        Compute memory-reconstructed node embeddings $\boldsymbol{Y}_j^{old}$,$\boldsymbol{Y}_j$, $\boldsymbol{Y}_{j+1}$ (subroutine 5)
        Compute target value using TD(0): $y_j = r_{j+1} + \gamma V_\theta(\boldsymbol{Y}_{j+1}, \theta_c)$, Eq.5.16
      end for
      for each transition $\xi_j \in \tau_i$ **do**
        Compute advantage $\hat{A}_j^{GAE}$, using Eq.4.7 and $V_\theta(\boldsymbol{Y}_j; \theta_c)$, Eq.5.16
        Compute probability ratio $r_j(\theta) = \frac{\pi_\theta(a_j \mid \tilde{\boldsymbol{o}}_{:j})}{\pi_\theta^{old}(a_j \mid \tilde{\boldsymbol{o}}_{:j})} = \frac{\pi_\theta(a_j \mid \boldsymbol{Y}_j; \theta_a)}{\pi_\theta^{old}(a_j \mid \boldsymbol{Y}_j^{old}; \theta_a^{old})}$, using Eq.5.14
        $\mathcal{L}^{ppo}$ += $\mathcal{L}_j^{CLIP}(\theta) + c_1 \cdot \mathcal{L}_j^{VF} + c_2 \cdot S_j(\theta)$ using Eq.4.9-4.12
      end for
    end for
    Perform gradient step to update θ using $\mathcal{L}^{ppo}$
  end for
  With validation frequency τ^{val}, save checkpoint of best model during training
end for

PPO implementation with GNN-LSTM latent feature extraction Training a model that includes an LSTM module using RL requires careful accounting of the hidden state of the LSTM. We will only consider a GNN-LSTM integration with the PPO algorithm and avoid the complications and potential instabilities that can arise when applying DQN to models that contain LSTMs [KOQ+19, HS15]. The PPO algorithm alternates between generating sequences (episodes) under the current policy and using this recent experience to perform a number of sequential policy gradient updates. There are various ways to approach the implementation. One is to keep a memory buffer of sequences and randomly sample minibatches such that stochastic gradient descent can be applied. However, hidden state accounting becomes vulnerable as re-used hidden LSTM states can become obsolete ('stale'), negatively affecting the quality of the gradient calculation. Moreover, if we want to use different graph classes and topologies during training, vectorizing the batch updates becomes very complex and error prone. This leads us to use a pre-set number of full episodes obtained during sequence generation to perform the gradient updates. Advantage values and probability ratios can be calculated for these complete sequences which means that we can accurately keep track of the

LSTM hidden states. The full PPO procedure using our GNN-LSTM latent feature extraction and parametrized Actor and Critic models is presented in Algoritm 3, with two helper functions specified in Algorithms 4 and 5. Further implementation details are offered in Appendix D.

Algorithm 4 Subroutine: generate PPO transition

1: Receive observation $\boldsymbol{o}_t = (\boldsymbol{X}_t, \boldsymbol{E})$ and LSTM state $\boldsymbol{Z}_{t-1}^{(out)}$, with $\boldsymbol{Z}_{:}^{(\cdot)} = (\boldsymbol{Y}_{:}^{(\cdot)}, \boldsymbol{C}_{:}^{(\cdot)})$
2: Set $\boldsymbol{Z}_t^{(in)} = \boldsymbol{Z}_{t-1}^{(out)}$
3: Calculate node embeddings, $\boldsymbol{H}_t = F(\boldsymbol{o}_t; \theta_{gnn})$
4: Apply the LSTM, $\boldsymbol{Z}_t^{(out)} = (\boldsymbol{Y}_t^{(out)}, \boldsymbol{C}_t^{(out)}) = R(\boldsymbol{H}_t, \boldsymbol{Z}_t^{(in)}; \theta_{lstm})$, Eq.5.13
5: Apply Actor network $\pi_\theta(v \mid \tilde{\boldsymbol{o}}_{:t}) = \pi(v \mid \boldsymbol{Y}_t^{(out)}; \theta_a)$, Eq.5.14
6: Take action $a_t \sim \pi_\theta(v|\tilde{\boldsymbol{o}}_{:t})$
7: Receive r_{t+1}, $e_{t+1} = a_t$, $\boldsymbol{u}_{t+1}$ from $\mathscr{I}$ (Eq.5.2), $\boldsymbol{o}_{t+1} = (\boldsymbol{X}_{t+1}, \boldsymbol{E})$, done$_{t+1}$ from Eq.5.3
8: Return transition $\xi_t = (\boldsymbol{o}_t, a_t, r_{t+1}, \boldsymbol{o}_{t+1}, done_{t+1}, \boldsymbol{Z}_t^{(in)}, \boldsymbol{Z}_t^{(out)})$

Algorithm 5 Subroutine: calculate memory-reconstructed node embeddings

1: Receive transition $\xi_j = (\boldsymbol{o}_j, a_j, r_{j+1}, \boldsymbol{o}_{j+1}, done_{j+1}, \boldsymbol{Z}_j^{(in)}, \boldsymbol{Z}_j^{(out)})$, with $\boldsymbol{Z}_{:}^{(\cdot)} = (\boldsymbol{Y}_{:}^{(\cdot)}, \boldsymbol{C}_{:}^{(\cdot)})$ the stored LSTM states
2: Calculate embeddings $\boldsymbol{H}_j^{old} = F(\boldsymbol{o}_j; \theta_{gnn}^{old})$, $\boldsymbol{H}_j = F(\boldsymbol{o}_j; \theta_{gnn})$, $\boldsymbol{H}_{j+1} = F(\boldsymbol{o}_{j+1}; \theta_{gnn})$, Eq.5.7
3: Calculate $\boldsymbol{Y}_j^{old} = R(\boldsymbol{H}_j^{old}, \boldsymbol{Z}_j^{(in)}; \theta_{lstm}^{old})$, $\boldsymbol{Y}_j = R(\boldsymbol{H}_j, \boldsymbol{Z}_j^{(in)}; \theta_{lstm})$, $\boldsymbol{Y}_{j+1} = R(\boldsymbol{H}_{j+1}, \boldsymbol{Z}_j^{(out)}; \theta_{lstm})$, Eq.5.13
4: Return $\boldsymbol{Y}_j^{old}$,$\boldsymbol{Y}_j$, $\boldsymbol{Y}_{j+1}$

Note that the PPO algorithm without an LSTM is recovered by simply removing the module in Fig.5.6 and setting $\boldsymbol{Y}_t = \boldsymbol{H}_t$ in equations 5.13-5.16.

5.4 Approach to training and evaluation

Diversity in graph datasets is useful when testing for generalizability of RL algoriths. We will define various ways to segment graph classes into train- validation- and test sets. For instance, one could train on subsets of small graphs and evaluate whether learned policies are effective on larger size graph. Or one could train on a limited amount of pursuers and evaluate how performance is affected when increasing the number of pursuers. We briefly introduce the approach to training and evaluation.

RL algorithms For basic RL experiments, the `Stable Baselines3` package was used. The `Pytorch Geometric` package was used to implement graph representation learning. However, for the more advanced experiments that combine GNNs and LSTMs, no general RL framework was sufficiently flexible to handle all scenarios we will be considering. Therefore, own implementations of DQN (Alg.2) and PPO (Alg.3) were written. References to the repository that contains the code used to conduct the experiments are included in Appendix D.

Training All our experiments involve the training of an escape agent using a sub-set of graph topologies as a trainset. The range of model architectures and hyperparameters used in this study is provided in Appendix D, Fig. D.5. Model sizes range from ca. 20k to 170k trainable parameters. The Adam algorithm [KB15] is used for all gradient based optimization. Dimensions of latent node embeddings and LSTM hidden states are kept below 64, which proved sufficient to demonstrate the concepts presented in this thesis. During training, model performance is both evaluated on the trainset itself (by averaging the return over a number of most recent rollouts, usually between 50-100) and on a validation set with held-out instances of the same graph classes incorporated in the trainset but with different initial conditions. Unless stated otherwise, a minimum of 3 seeds is used, which is increased up to 10 seeds if high variance of performance is observed. For training, Nvidia 1080Ti and Titan RTX GPUs nodes were used from the Lisa compute cluster. Models that incorporate LSTM modules, trained using larger graph classes (up to 1,000 nodes) require up to 20Gb of GPU VRAM memory. Training times vary between 2-12 hours depending on settings.

Evaluation Our trained models will be evaluated using all instances with pre-calculated pursuer responses of selected graph classes that were not part of the trainset. We will use two metrics:

$$\begin{aligned} \text{Average Return per Episode (ARPE)} &= \frac{\text{Accumulated discounted rewards over all episodes}}{\text{Total number of graph instances evaluated}} \\ \text{Success Rate (SR)} &= \frac{\text{Number of episodes in which escaper reaches a target node}}{\text{Total number of graph instances evaluated}} \end{aligned} \tag{5.17}$$

ARPE is used for model validation and can distinguish between different routes to target nodes. SR looks at the outcome, irrespective of the efficiency of the route. ARPE is the standard metric used in RL and directly related to the reward function that forms the agent's objective. However, is hard to compare performance test results across different graph classes with ARPE, as the maximum attainable return will vary between these. SR is used when comparing performance between different graph classes. ARPE and SR correlate well but ARPE is a more exact metric. For PPO, deterministic evaluation (using only actions with maximum assigned probability) is used unless the stochastic policy performs better. Performance is compared to a 'Collision Risk Avoidance' heuristic described below.

Benchmark performance using heuristics Basic heuristics for escape behaviour include shortest path- or minimum degree path algorithms. Traversing nodes with low degrees could be advantageous as one could expect search- and pursuit agents to prefer highly intersected regions. However, both methods perform poorly on our test scenarios. In order to obtain a more challenging performance benchmark, a discrete-time 'Collision Risk Avoidance' (CRA) heuristic was developed, inspired by [SO11]. The basic idea of CRA is that every time a pursuit agent i is observed on a node $u_t^{(i)} = v$, new probability mass is uniformly assigned to all neighboring nodes in $\mathcal{N}_v$ to represent this pursuer's potential location in the next timestep. Any existing probability mass present on the graph from previous timesteps is propagated using the same assumption of random movement. Once all probability mass is distributed, edge weights are assigned based on the resulting target node risks. Then, the lowest cost Dijkstra path to any of the target nodes in $\mathcal{T}$ is used to direct the escaper. This approach will also work under partial observability. However, it is impossible to remove probability mass from prior sightings as one cannot know if an observed pursuer at time t is the same as one observed on a neighboring node at $t-1$. The algorithm still works, however, as it is the relative accumulated node risk that informs risk avoidance decisions. Note that this algorithm bears some resemblance with the notion of message passing in GNNs, but as a scripted (not learned) procedure. The distribution of target node risks could be seen as a latent feature on which decisions are based.

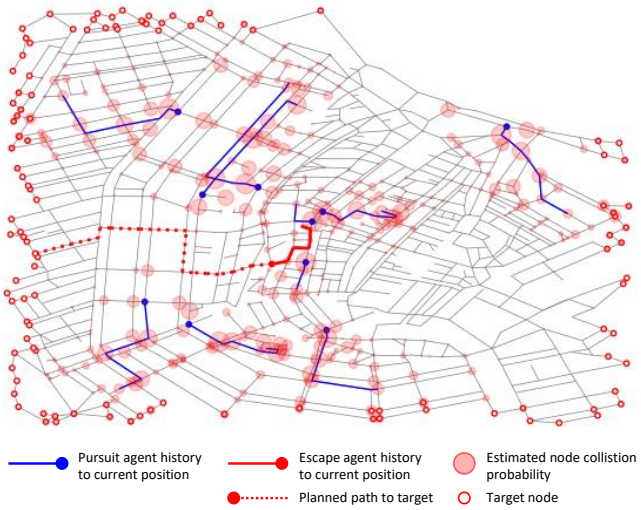


Figure 5.7: Example frame (10 pursuers, t=5) of a rollout on the AMS graph using the CRA heuristic

Although CRA generalizes very well, it has several disadvantages. First, it is unable to predict and exploit any pursuer patterns; it just assumes uniform action distributions. Second, outdated node risks are not removed from the graph these can interfere with future decisions. Lastly, CRA has a time complexity that scales poorly with size and length of episodes. The algorithm runs in $O\left[|\mathcal{V}| * d_{max} + |\mathcal{T}|(|\mathcal{E}| + |\mathcal{V}| \log |\mathcal{V}|)\right]$ and will get slow for larger graphs as an episode progresses (more and more non-zero node risks need to be propagated). Every timestep, all node probabilities need to be distributed and subsequently, a Dijkstra shorest path needs to be calculated for all target nodes. Nonetheless, CRA is versatile and sets a good benchmark as a basic heuristic. Figure 5.7 shows CRA in action, with the escaper clearly avoiding regions of higher perceived risk.

Chapter 6

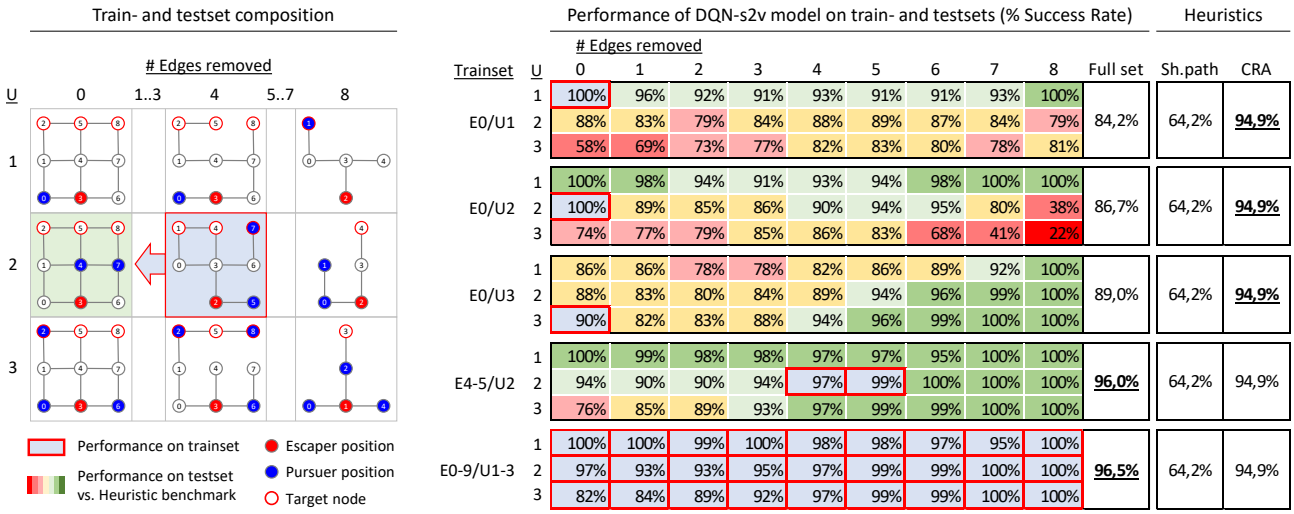
Experiments

We take a iterative approach to developing our method for solving the Search & Pursuit-Evasion (SPE) task under partial observability. The reason for this is that several choices in the escape agent's model architecture are not straightforward and require experimentation. Particular challenges are faced when scaling up the environment to larger graphs and when limiting the observability level of the escape agent. As a first step, we investigate whether models that use GNNs to extract latent node features can be trained using RL to succeed in the SPE task on small graphs and under full observability. We test whether these trained models can generalize to out-of-distribution graphs. Second, we investigate whether these models can handle larger graph classes, with up to 2,500 nodes. Once this is confirmed, we extend the approach to include partial observability and simulate the scenario where the escaper only has sporadic information updates on pursuer positions. We push the performance boundary as far as we can. Lastly, we investigate several aspects of the trained model's behavior, including the influence diversity in the trainset, the model's predictive capability and its effective planning horizon.

6.1 Generalizability over graphs and number of pursuers

Our first experiment is designed to verify whether our trained models can generalize to (i) other (unseen) graph topologies, and (ii) to graph instances that are populated with varying number of pursuers and their initial positions. We assume full observability for now and start with composing trainsets by selecting certain sub-sets (segments) from the Manhattan 3x3 graph permutation dataset (see sections 5.2, C and Fig.C.2). We then train the agent using all instances (i.e. different initial positions) of the graph topologies contained in the trainset. Performance of the trained agent is evaluated on all graph sub-sets that were excluded from the trainset. The left panel of Figure 6.1 illustrates this approach to train- and testset composition. In this example, the trainset (center square with the red border) consists of all graph instances that are possible with four edges removed from the full Manhattan 3x3 topology and with two pursuit agents randomly placed at the start of an episode. We denote this subset as 'E4/U2'. Performance is tested on the segment 'E0/U2' (in green, to the left) that contains all full Manhattan 3x3 topologies, also with two pursuit agents present. Note that we always initialize the escape agent on the bottom center node and the nodes in the top row are always the escaper's target nodes. Instances are solvable if there exists an escape route, given initial positions, that avoids capture and reaches one of the target nodes. We have pruned all datasets to include only solvable instances.

This is clearly a very basic task that can be completed in at most 5 timesteps. However, the task does require some maneuvering and an ability to find optimal (shortest) paths to target nodes while avoiding pursuers. As we will see, the modeling approach described in section 5.3 enables an agent to learn these capabilities using a 4-dimensional node feature definition NFM_4 (see Fig.D.2) that contains information on the escaper's current position, the nodes it has visited before, the target nodes and the current positions of the pursuers. We define a Q-network (Eq.5.8) with a *struc2vec* GNN (Eq.4.17) to extract node embedding using a latent dimension d'=64 and τ_{max}=3 propagation steps. We train this model, that has ca. 30k parameters, using DQN (Alg.2) with e_{max}=2,500 episodes and



Performance of DQN-s2v model on train- and testsets (% Success Rate)

Trainset	U	# Edges removed 0	1	2	3	4	5	6	7	8	Full set	Heuristics Sh.path	CRA
E0/U1	1	100%	96%	92%	91%	93%	91%	91%	93%	100%	84,2%	64,2%	**94,9%**
	2	88%	83%	79%	84%	88%	89%	87%	84%	79%			
	3	58%	69%	73%	77%	82%	83%	80%	78%	81%			
E0/U2	1	100%	98%	94%	91%	93%	94%	98%	100%	100%	86,7%	64,2%	**94,9%**
	2	100%	89%	85%	86%	90%	94%	95%	80%	38%			
	3	74%	77%	79%	85%	86%	83%	68%	41%	22%			
E0/U3	1	86%	86%	78%	78%	82%	86%	89%	92%	100%	89,0%	64,2%	**94,9%**
	2	88%	83%	80%	84%	89%	94%	96%	99%	100%			
	3	90%	82%	83%	88%	94%	96%	99%	100%	100%			
E4-5/U2	1	100%	99%	98%	98%	97%	97%	95%	100%	100%	**96,0%**	64,2%	94,9%
	2	94%	90%	90%	94%	97%	99%	100%	100%	100%			
	3	76%	85%	89%	93%	97%	99%	99%	100%	100%			
E0-9/U1-3	1	100%	100%	99%	100%	98%	98%	97%	95%	100%	**96,5%**	64,2%	94,9%
	2	97%	93%	93%	95%	97%	99%	99%	100%	100%			
	3	82%	84%	89%	92%	97%	99%	99%	100%	100%			

Figure 6.1: Model performance vs. heuristics, for different compositions of train- and testsets using Manhattan 3x3 graph permutations

a replay memory buffer capacity of m=2,000 transitions. Test performance is presented in the right panel of Fig.6.1. For this basic task, a single seed is used for each trainset.

Some interesting behavior van be observed. When training with data from a single full Manhattan 3x3 segment ('E0/U1', 'E0/U2' or 'E0/U3' in Fig.6.1), average performance on unseen graph instances reaches between 84-89%. This is better than a simple shortest path heuristic (which achieves ca. 64%) but it does not match the Collision Risk Avoidance (CRA) heuristic. However, if we train on a sub-set of two diverse segments (segment 'E4-5/U2' contains nearly 750 graph topologies and over 4,300 graph instances) performance improves for most unseen segments and the overall performance on the full graph dataset jumps to 96%. Here, we see evidence that this model can successfully learn this task and can generalize to distributions of graph instances outside that of the training data. To illustrate typical learned behavior, an example rollout is shown in Appendix A, Fig.A.1a).

Note that a maximum performance of 96,5% is reached when the full dataset of this graph class is used for training. This is not surprising given that the agent's experience is now based on the most diverse set of graph instances. Whether this model is able to perform on larger graphs is investigated next.

6.2 Scale-up to real-world road networks

We are interested in the performance of models trained using sub-sets of small graphs, evaluated on sub-sets of larger (unseen) graphs. We immediately run into a limitation. When using the 4-dimentional node feature definition NFM_4 (see Fig.D.2) from our first experiment, the escape agent will not be able to effectively navigate larger graphs. The signal that contains information on target node status $i_w^{(\mathcal{T})}$ (Eq.D.3), as part of the feature $\boldsymbol{x}_w$ of some target node $w \in \mathcal{T}$, needs to 'reach' the escape agent, otherwise the escaper lacks crucial information on its objective. Recall, we defined policy- or value function approximators based on a transformation of latent node features $\boldsymbol{h}_i$, $i \in \mathcal{V}$ (Eq.5.8-5.11). These node embeddings are created by applying a pre-defined number of k GNN layers. This will give each node a receptive field of all nodes reachable within k hops. The escaper, at position $u = e_t$ will base its decision on a transformation of the node embeddings of its neighbors, $\boldsymbol{h}_v$, $v \in \mathcal{N}_u$ and the mean of all node embeddings in the graph, $\bar{\boldsymbol{h}}$. If target nodes are more than k hops away, the escaper will not receive this signal and will have a hard time finding paths to those target nodes. We want to avoid increasing the number of GNN layers, as that will require more computation and may introduce bottlenecks in the information propagation, see section 4.2.

We remedy this caveat by adding two elements to the feature definition: $i_u^{(d\mathcal{T})}$ (Eq.D.4, a normalized option-to-target value: nodes that are close to many target nodes will be assigned a higher value)

and $i_u^{(aT)}$ (Eq.D.5, absolute distance to target: a node is assigned the minimum number of steps to the nearest target node). To further assist navigation, we encode static or dynamic movement of observed pursuit agents in the node features: $i_{v,t}^{(ps)}$ (Eq.D.7, the count of static pursuers on a node that have not moved in the last timestep) and $i_{v,t}^{(pm)}v$ (Eq.D.8, the count of dynamic pursuers on a node that have moved in the last timestep). We will use the resulting 7-dimensional node features NFM_7 (D.2) from now on.

With these adjustments, we are ready to conduct our scale-up experiment. In Appendix E1, results are presented of detailed experiments that use different variations of testset and trainset compositions. Here, we summarize one specific set-up that shows an important result. We use two RL algorithms (DQN and PPO) and train two escape model variants: one that employs a *struc2vec* GNN and one that employs a *GATv2* GNN. We limit model complexity and keep their sizes under 30k parameters. Details on model definitions and hyperparameters are provided in Appendix D. Each of these variants is trained under full observability for the escape agent, using the AMS trainset that represents a 2km ×2km inner city road network of Amsterdam with 975 nodes. Instances in this trainset place the escape and pursuit agents at random initial positions and assign target nodes either along the outer border of the graph (113 nodes in total), or at random (between 10-20 random nodes). Further details on this trainset are provided in Appendix D. The progression of performance and loss during training for the *GATv2* model variant is shown in Fig.6.2. Note that for PPO, the weight c_2 of the negative entropy loss component $\mathcal{L}^S$ (see Eq. 4.10) is set to zero and hence $\mathcal{L}^S$ is not minimized, leading to low-entropy (i.e., deterministic) action distributions.

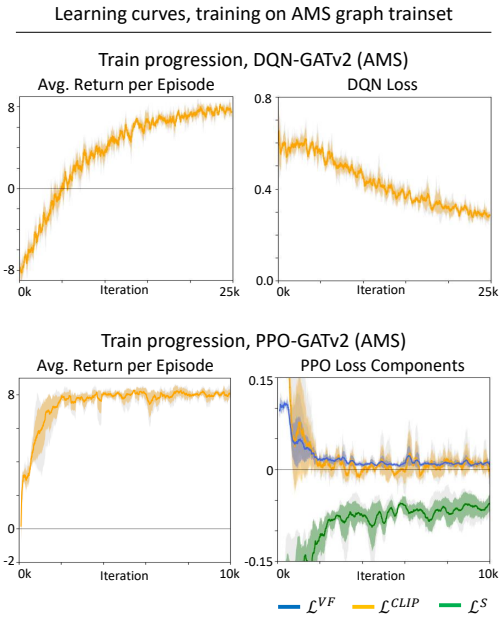


Figure 6.2: Selected learning curves

Performance of model variants, trained on AMS graph trainset (% Success Rate)

	DQN				PPO				Heuristics			
	struc2vec[1]		*GATv2*[1]		*struc2vec*[1]		*GATv2*[1]		CRA		Sh.path	
testset	SR	std	SR	std	SR	std	SR	std	SR	std	SR	std
M3	31%	11%	50%	8%	77%	8%	65%	10%	**95%**	-	64%	-
M5	16%	13%	60%	11%	55%	7%	53%	14%	**84%**	-	48%	-
Metro	19%	12%	84%	8%	64%	1%	86%	9%	**93%**	-	62%	-
AMS[2]	91%	2%	**93%**	1%	61%	1%	89%	6%	88%	-	49%	-
UTR	82%	2%	**92%**	3%	51%	2%	83%	11%	79%	-	46%	-
ROT	57%	24%	94%	1%	78%	2%	**95%**	2%	91%	-	73%	-

Notes
Performance shown represents the success rate (%) and standard deviation over 5 seeds
Trainset, testsets and model variants are described in Appendix D
1. Models variants used are DQN.s2v.C, DQN.GATv2.A, PPO.s2v.D, PPO.GATv2.B (see Appendix D)
2. AMS testset contains graph classes that were part of the AMS trainset (see Appendix D)

Color coding for escape agent RL performance
xx% better than CRA heuristic
yy% between shortest path and CRA heuristics
zz% worse than shortest path heuristic

Figure 6.3: Results, scalability experiment

At this stage, we are only interested in a proof of concept of our approach: can our models outperform the CRA heuristic at all. To test this, we evaluate performance of the trained models on smaller unseen graph sets (testsets M3, M5, Metro) and larger unseen graph sets (testsets UTR and ROT, two other inner city graph topologies). The resulting success rates (average and standard deviation over 5 seeds) are shown in Fig.6.3. Several observations can be made:

- Training solely on instances of the AMS graph topology transfers to other large graphs (UTR, ROT). It transfers to some extent to smaller topologies (Metro, M5, M3) although the performance of the CRA heuristic cannot be matched on these small and very local evasion tasks.
- The *struc2vec*-based GNNs cannot match *GATv2*'s performance and we will discard them. The *struc2vec* expressiveness cannot be expanded straightforwardly, contrary to *GATv2* (currently only two attention heads are used and weights are shared between *GATv2* layers).

- Appendix E1 shows that expanding trainset to include multiple graph classes enables the *GATv2*-based models to improve performance across the entire range of testsets.
- Experimentation with increasing the number of GNN layers k (Appendix E1) shows that test performance is negatively affected when k increases beyond 7. We will keep $k = 5$ for the remainder of our experiments.

The conclusion that we can draw from this experiment is that our approach is feasible, but needs to be expanded such that the escaper model can operate under a more realistic partial observability scenario.

6.3 Extension to partial observability

We have arrived at the full scope of the challenge posed in our research question: can an RL escape agent learn to exploit an unknown pursuer strategy over a wide range of graph instances and under partial observability? For reasons described in section 5.3, we limit our experiment to the PPO algorithm when incorporating a memory state in our modeling approach. Furthermore, given the results from the previous section, we work exclusively with the *GATv2* GNN model variant. To keep our experiment most realistic, we train a model using the AMS trainset, and evaluate on test graphs UTR.F and ROT.F, that contain instances of other large graph topologies with different statistics such as number of nodes, average weighted node degree and density. We keep the number of pursuit agents constant at U=10 on all graphs and let them follow pursuer strategy $\mathscr{I}_{base}$ (see Eq.5.2).

We apply the concept of latent node feature reconstruction using memory (see Fig.5.6). In Appendix E2, we test our implementation on a small example graph and experiment with different positions of the LSTM module. This test shows that the best position of the LSTM is when it acts directly on the node embeddings (position 2 in Fig.E.3). The resulting memory-reconstructed node embeddings can then be passed on to both the Actor and Critic networks. The test also shows that frame-stacking (see Sec.4.1) offers a viable and simpler alternative to keep track of previous observations, and we will include technique in our experimental configurations. We define a baseline PPO-GATv2-LSTM model using Eq.5.14-5.16 with latent dimension $d' = 64$, $k = 5$ *GATv2* layers, each with 4 attention heads and unshared weights, and an LSTM module with one layer and a hidden dimension of $r = 64$ (a full model- and hyperparameter specification is provided in Fig.D.5). This model has ca. 170k parameters. We need to specify the level of visibility during training according to Eq.5.6. For our baseline model, we offer the escape agent the opportunity to learn over a spectrum of visibility levels and sample the observation probability $p_{vis}^{(train)} = p_{train} \sim \mathcal{U}_{[0.5,1.0]}$ for each graph instance used during training. The resulting train progression (performance and loss components) is shown in Fig.6.4. Note that both the performance and clipped surrogate loss curves progress with a higher variance during training than the previous experiment. This is because graph instances and visibility levels are sampled and will be different for each training iteration, affecting the attainable rewards. Like before, we exclude the entropy loss component $\mathcal{L}^S$ in the PPO loss function. The entropy loss curve settles at a lower level compared to the training runs with full visibility (Fig.6.4), indicating that the model's predicted action probabilities have become less deterministic. This is as expected, given the loss of information under lower visibility. The test performance of the different model configurations is shown in Fig.6.5. On the horizontal axis, visibility during testing decreases from full visibility (p_{vis}=1.0) to zero visibility. The left panel shows performance on the AMS.F graph topology, which was part of the trainset. The center and right panels shown performance on UTR.F

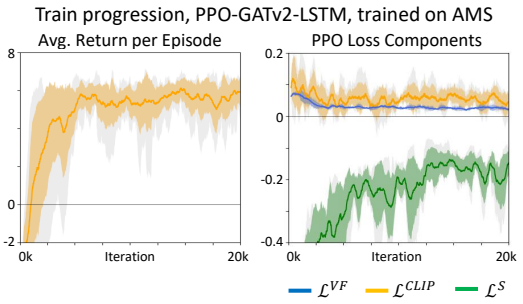


Figure 6.4: Learning curves under mixed observability, $p_{train} \sim \mathcal{U}_{[0.5,1.0]}$

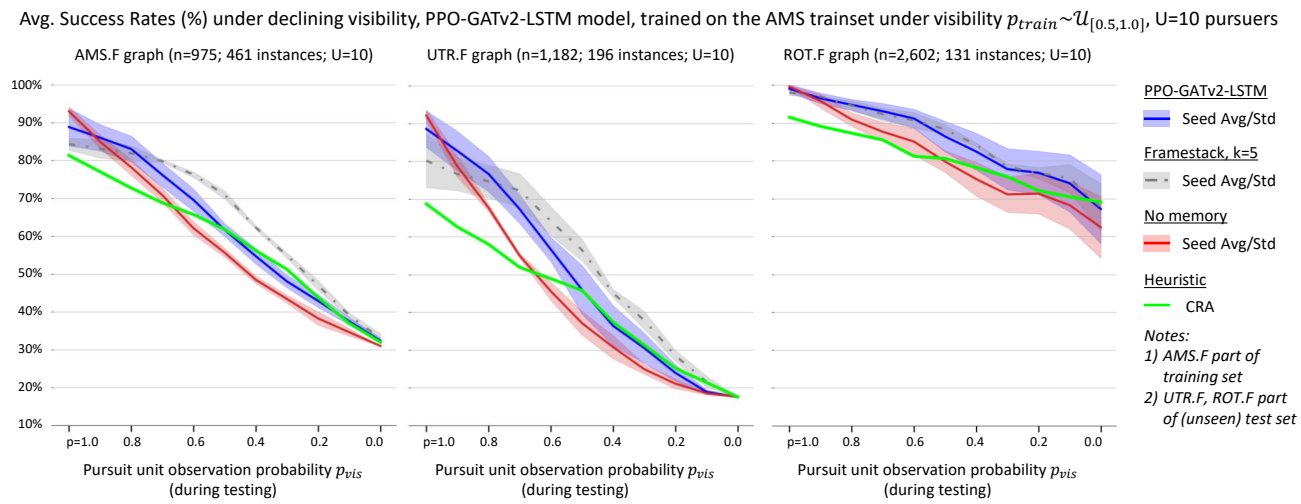


Figure 6.5: Test performance under partial observability of the baseline PPO-GATv2-LSTM model

and ROT.F, both of which are larger graph topologies that were not part of the trainset. For reference, an illustration of typical escape behaviour on AMS.F under different levels of visibility is shown in Fig.A.2 in the Appendix. Three model configurations are compared to the CRA heuristic: i) the baseline model with an LSTM; ii) the baseline model without LSTM but using frame-stacked node features (with k=5 most recent frames as memory); and iii) the baseline model without an LSTM. We make the following observations:

- The CRA heuristic performance profiles give an indication of how difficult, on average, the task of escaping on the three graph topologies is. CRA scores highest on ROT.F, which apparently is the easiest graph to navigate[1].
- The PPO-GATv2 model without memory (the red line in Fig.6.5) outperforms CRA in the region of high visibility, but deteriorates fast and drops below the benchmark when visibility is reduced to under p_{vis} <0.7.
- Both memory-endowed models (LSTM- and framestack configurations) outperform CRA down to a visibility level of around $p_{vis} \approx 0.5$ on all three graph topologies. The performance benefit is highest on UTR.F, which is also the most difficult graph to navigate according to CRA.
- The configuration that uses frame-stacking performs best overall, both in terms of success rates and variance across seeds. However, when adding more pursuit agents to the graph, the performance of this configuration cannot sustain its advantage over the CRA heuristic, as can been seen in Fig.6.6 (more details can be found in Appendix E). This might be explained by the fact that the LSTM module benefits from the ability to sustain longer memory sequences, and better account for the larger number of sporadic sightings of pursuers over time.

The performance curve of the PPO-GATv2-LSTM model, trained under mixed visibility, is the highest we were able to achieve. The actual performance limit is unknown to us. Note that we base our conclusions on the Success Rate metric (Eq.5.17), which allows for better comparison across different graph sizes. Test results expressed in Average Return per Episode (ARPE) is shown in Fig.E.8 in Appendix E5. The metrics correlate well, relative performance patterns are equivalent.

6.4 Detailed performance analysis

Observing the escape agent's behavior in the previous sections begs several explorative questions. We touch upon four of the most prominent.

[1] upon inspection of successful escape routes, going South from the center node in Rotterdam B.3 across the Erasmusbrug appears to be hard to intercept due to parallel lanes and multiple available short-cuts

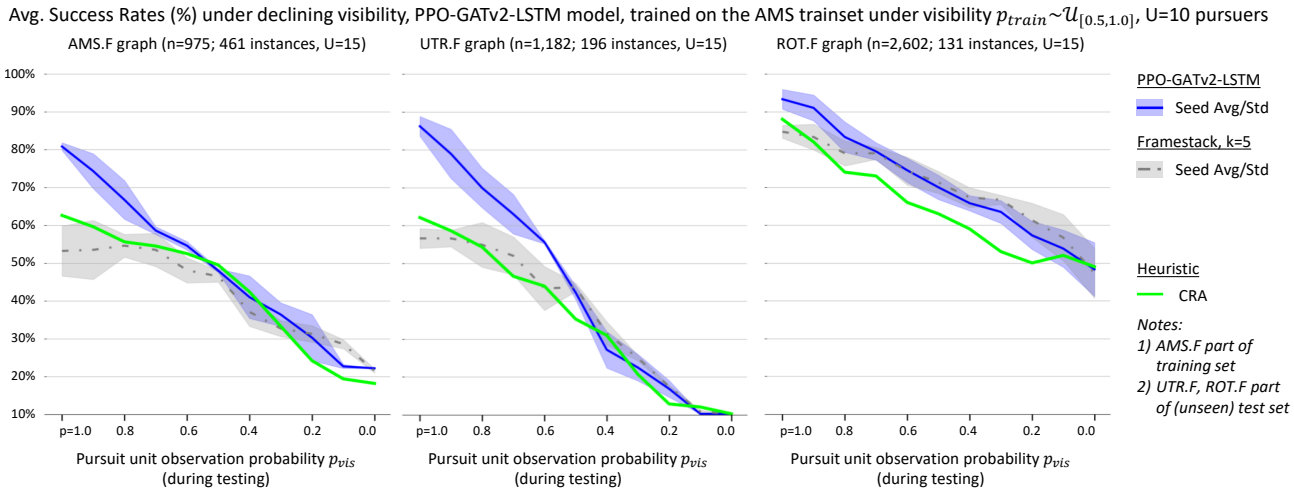


Figure 6.6: Test performance of the baseline PPO-GATv2-LSTM model with U=15 pursuit agents

1. Influence of visibility level during training The performance of the baseline PPO-GATv2-LSTM model under partial observability, shown in Fig.6.5 was achieved by sampling visibility probabilities during training from $p_{vis} = p_{train} \sim \mathcal{U}_{[0.5,1.0]}$. The reasoning behind this choice was that such a mix of experiences could assist in learning to deal with limited visibility. It was assumed that expanding this range to lower levels of visibility would not add much useful experience given the lack of information to inform useful actions. Could a better performance be achieved for a given visibility level if training was performed under that same fixed ('pointwise') visibility level? To answer that question, we train our baseline model using the AMS trainset and specific fixed visibility probability. We pick two scenarios: $p_{train} = 0.9$ and $p_{train} = 0.5$. We evaluate these trained models on the AMS.F graph topology, over the full visibility spectrum. The result is shown in Fig.6.7. The performance curve for $p_{train} = 0.9$ starts off higher in the region of high visibility, but drops quickly as visibility is reduced. Interestingly, the performance curve for $p_{train} = 0.5$ shows comparable performance under high visibility. It sustains performance slightly better in the low visibility region, but it cannot match the CRA heuristic, even at the visibility level under which is was trained. In contrast, the original baseline model trained under the broader visibility range can keep up performance over the entire range. Indeed, exposure to more diverse visibility levels proves to be useful for this task.

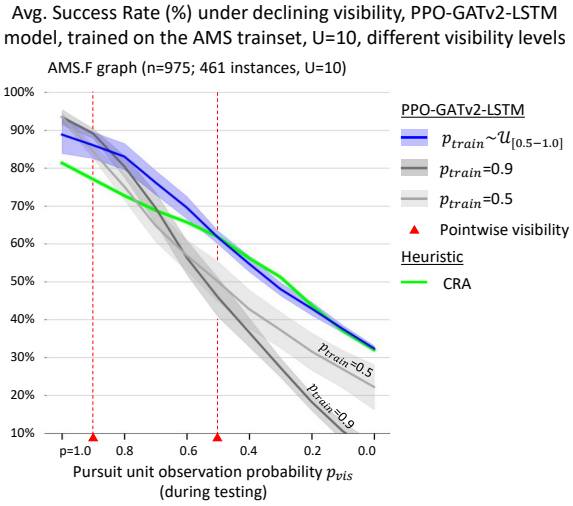


Figure 6.7: Performance comparison, pointwise vs. range-based visibility levels

2. Predictive capability The promise of using Graph Representation Learning in combination with Reinforcement Learning for the Search & Pursuit-Evasion problem on graphs was that with such a technique, a model would be able to learn the heuristics required to solve the task. This is indeed the case: we showed that without pursuit agents present, the model can learn to find shortest paths to target nodes, and with pursuit agents present, it can execute evasive actions and navigate to target nodes without being captured. But does the escape agent merely learn some generic escape maneuvers, or does it actually learn characteristics of the pursuer strategy that it can exploit? To test this, we create a graph state in which the escaper's best action requires a prediction of the pursuer's response. Consider the permuted Manhattan 3x3 graph in Fig.6.8 with one pursuit agent present. This represents a graph topology that was kept outside the trainset.

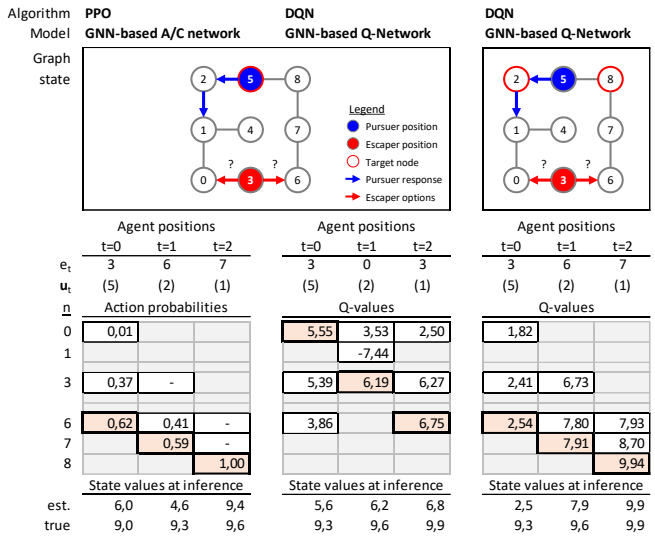


Figure 6.8: Decisions by trained agents on unseen graphs (requires prediction of opponent's action)

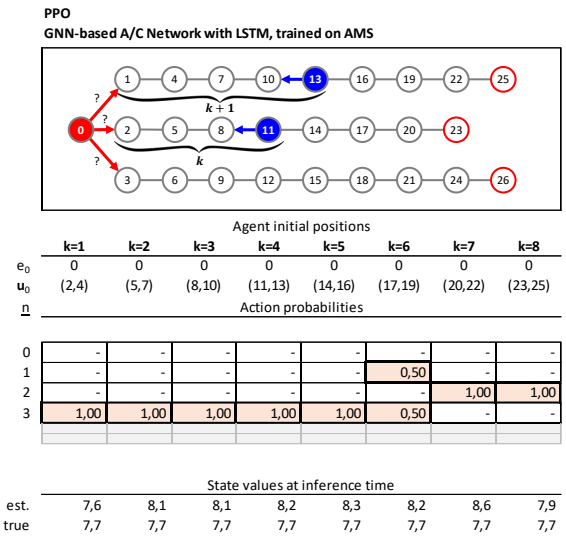


Figure 6.9: Longer range signal detection (limited by GNN's receptive field)

The pursuer response function $\mathscr{I}_{base}$ (Eq.5.2) on which this escape agent was trained will direct the pursuer from its initial position on node 5 to node 1; this is the highest degree node associated with the highest probability of intercepting an undirected random walker. When inspecting the action probabilities that the trained model outputs (the bottom left table in Fig.6.8) we see that indeed, this highest probability is assigned to choosing node 6, followed by node 7,8 to target node 5. This is evidence of predictive capability by the escape agent. It has learned to read and interpret a new, unseen graph state and infer that the pursuer will go left and hence choose right as the best action in this case. This behavior, however, appears to be fragile. A Q-network trained using DQN does not select this first action (center table in Fig.6.8), but first walks to node 0 before changing direction to avoid capture. A slight adjustment in target node designation (right table) leads the same Q-network to predict the correct best action. The Q-network is a deep graph neural network which makes it very hard to interpret these results and explain how the model arrives at them.

Going back to our key result visualized in Fig.6.5, one could argue that the fact that the baseline PPO-GATv2-LSTM model outperforms the CRA heuristic in some part of the visibility region along the x-axis, is evidence of predictive capability. How else would the agent be able to do better than the uniform distribution of node risk that CRA uses? In order to outperform the CRA benchmark, the escape agent will have to learn more than merely basic avoidance tactics. It learns to predict, to some extent, pursuers' responses in arbitrary graph states. If this is true, what would happen if the pursuers change their strategy? One would expect that the performance advantage resulting from the predictive capability is lost. We investigate this next.

3. Changing pursuer strategy In section 6.3, we trained a baseline PPO-GATv2-LSTM model to dictate the escaper's behavior. During training, a pursuer strategy (or: response function) $\mathscr{I}_{base}$ was used. This resulted in a test performance of the escaper shown in Fig.6.5. If we change the pursuer response function at test time to $\mathscr{I}_{alt}$ (the nature of this change is described in Appendix B), the baseline escape agent's test performance will be affected. This impact is shown in Figure 6.10. There are several observations to be made.

- The performance curve for the CRA heuristic benchmark under $\mathscr{I}_{alt}$ moves up compared to its performance under $\mathscr{I}_{base}$. This holds for all three graph topologies shown, as long as p_{vis} >0.5. This means that the alternative pursuer strategy $\mathscr{I}_{alt}$ is easier to beat by an escaper under high visibility than the baseline strategy.

- Both on the graphs AMS.F and ROT.F, the baseline escape model, trained on $\mathscr{I}_{base}$ (blue line) can barely outperform the CRA benchmark when tested under $\mathscr{I}_{alt}$. The escaper has lost some

of its predictive capability following the same reasoning as before.

- In order to perform a counterfactual test, we retrain the escape agent under the new pursuer response $\mathscr{I}_{alt}$. As can be seen in the figure (orange line), the escape agent's performance increases again, confirming our hypothesis. If the escape agent is able to learn to predict the movement of pursuers on a given graph state, it can improve performance by exploiting this information. For completeness, the inverse scenario (train on $\mathscr{I}_{alt}$, test on $\mathscr{I}_{base}$) is shown in Fig.E.7 in Appendix E4. A similar effect is observed.

- Interestingly, on the UTR.F graph, the baseline escape model trained on $\mathscr{I}_{base}$ still does relatively well when tested on $\mathscr{I}_{alt}$ compared to the CRA heuristic. However, the success rate of the escape model trained under $\mathscr{I}_{alt}$ improves by a margin of up to 25% in the medium visibility range.

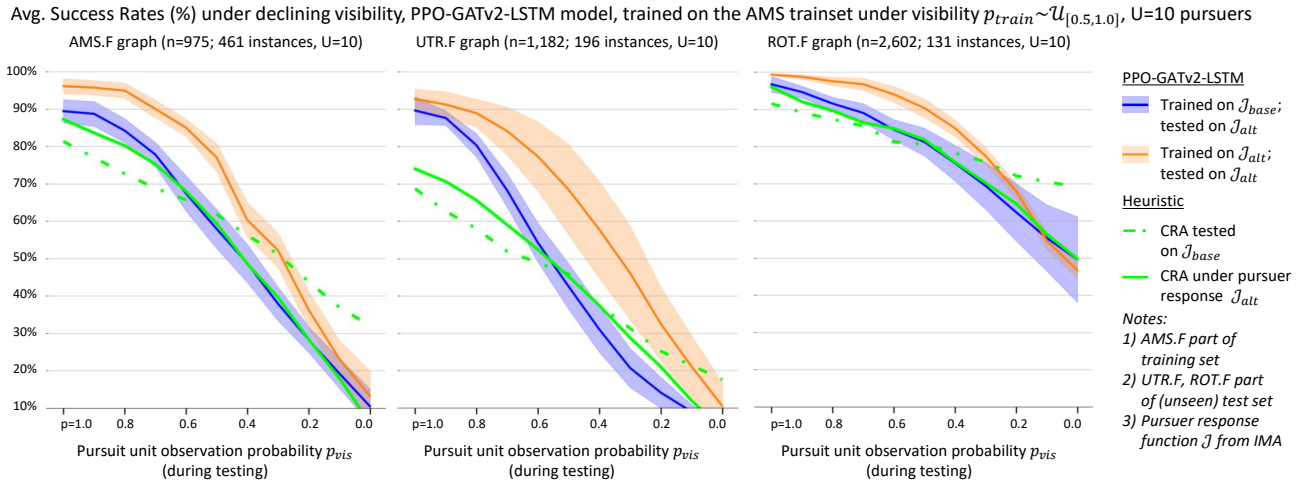


Figure 6.10: Impact of varying the pursuer response function $\mathscr{I}$ on test performance

4. Planning horizon Our last inquiry investigates the amount of planning and foresight that our escape agent demonstrates. In order to test this, we evaluate the trained PPO-GATv2-LSTM agent on another unseen test graph, shown in Figure 6.9, with two pursuit agents present. The escape agent starts on node 0 to the left and can choose between three tracks that lead to target nodes. The path through the middle is the shortest, but it is blocked by a pursuit agent. We vary the initial positions of the two pursuers by setting the value k as shown in the graph. The table below the graph shows the action probabilities to each of the reachable nodes for the escaper at $t = 0$.

We see that as long as the middle pursuit agent is nearby, the correct best action (node 3) is selected with certainty. As k is increased, the signal detection horizon required to sense the presence of a pursuit agent on the graph increases. At $k = 6$, the escaper loses the ability to detect the upper pursuit agent (who starts on node 19 in this case), but still senses the middle pursuer (on node 17 in this case) and assigns equal probabily to selecting the top or bottom routes. For $k \geq 7$, the escaper is not able to detect any pursuit agent and chooses the shortest middle route, assuming it poses no risk of capture. During the rollout that follows this inital action, the escaper will backtrack as soon as a pursuer enters its receptive field.

We have demonstrated a major, and to be expected, shortcoming of our modeling approach. Because the graph representation is learned using a GNN with (in this case) 5 layers, the message passing operation can only convey node features from at most 5 hops away. We ran into a similar problem earlier (section 6.2) when we required the escaper to be able to perform long-range pathfinding, and solved it by extending the node feature definition with additional graph information. However, using feature engineering to solve the limited detection range of pursuers that engage in dynamic behavior is undesirable. It is unrealistic to assume that an escape agent can keep track of such additional accounting. Still, the above suggests that the performance boundary can be pushed up further if longer range planning capability is added to the escaper model.

Chapter 7

Discussion

The results from our experiments show that a well designed parametrized policy function can be trained such that it demonstrates sophisticated escape behavior under partial observability. This behavior generalizes to larger and unseen graphs and to an arbitrary choice of initial conditions on these graphs, such as the position of escape and pursuer agents and the set of target nodes for the escaper. While the resulting escape performance improvement over a scripted heuristic Collision Risk Avoidance (CRA) algorithm may not seem overwhelming, it is a valuable result. The model is able to learn the heuristics necessary to define best actions, instead of having to rely on human-engineered heuristics and scripted algorithms that have to account for all possible graph structures and edge cases that can be encountered.

A major limiting factor of this approach is the bounded planning horizon that can be expected from the escaper model. The amount of layers in a Graph Neural Network defines the number of hops over which node features (or in their transformed form: messages) can be propagated. This sets an upper bound on the receptive field of individual nodes. Crucial information such as pursuer presence or target node status can only reach the escape agent if it is within its receptive field. It is not straightforward to increase this receptive field, due to computational stability and cost. When the number of layers in a GNN grows too large, an effect identified as 'oversquashing' [AY21] can hinder the network's stability and effectiveness. As a reference, our baseline model uses 5 layers and our largest test graph has ca. 2,600 nodes and a maximum distance between nodes of 65. We expect further performance gains if longer planning horizons can be achieved.

Based on our results, we can suggest a performance metric that can be used to evaluate a particular pursuer strategy. The area under the performance curve of a trained escape model is an indicator of how easy it is to beat a certain pursuit strategy. One has to be careful though; this area depends on graph topology, selection of the escaper's target nodes, number of pursuers and the way in which initial conditions are sampled. One could define a protocol in which a range of scenarios are covered, as well as a fixed escape agent model with some training regime and computation budget. Different pursuer strategies could then be evaluated and compared based on the resulting performance curves of the escape agent. We already saw an example of such an evaluation: the baseline pursuer response function $\mathscr{I}_{base}$ performs better than an alternative formulation $\mathscr{I}_{alt}$. The quantification of this difference follows from the area under the escape performance curves when opposing $\mathscr{I}_{alt}$ (Fig.6.10), compared to the area under the escaper performance curves when opposing $\mathscr{I}_{base}$ (Fig.6.5). Simulations using a trained escape agent model can potentially help improve a given pursuer strategy. For instance, these simulations offer statistics on (unexpected) evasion routes and advantageous initial pursuit unit distributions. We observed that when starting from the center of the ROT graph (the inner city graph of Rotterdam) an escaper can benefit from an efficient southern route to a set of border nodes. Knowing this, it might be beneficial to have a surveillance policy that ensures that there is always a police unit that covers this southern district.

Throughout this work, a number of modeling choices had to be made while formalizing the complex

and unpredictable nature of real-life escape scenarios. A fair question to ask would be whether a real human escaper would ever demonstrate the type of learning and behavior that our escape model develops. This is hard to answer but surely highly doubtful. Yet, it is not inconceivable that a sophisticated and well-prepared escaper can acquire sporadic information updates on pursuer positions and actions. Based on this, it might maintain a mental model of the world around him/her (the human equivalent of a latent representation) and make (possibly intuitive) decisions based on this. Note that the objective of this work was not to mimic human behavior. The purpose was to investigate to what extent a pursuer strategy can be exploited, without explicitly knowing this pursuer strategy. We succeed in demonstrating that this is indeed possible, as shown in Figures 6.5 and 6.10.

Future work Based on the conclusions of our experiments, a number of improvements could be pursued that address weaknesses in the approach that we took:

- The approach we took suffers from an inherent limitation of the escaper to engage in long-range planning. This should be addressed and there are multiple potential improvements to be considered. Incorporating Monte Carlo Tree Search in the approach would be an obvious extension to consider first. Another approach would be to define a value- or policy function not as a transformation of node embeddings of neighboring nodes (Eq.5.8,5.9), but of embeddings of entire routes to target nodes. Such a technique was used in [YWS+19], albeit on simpler tasks.

- We had to make various limiting assumptions in our modeling approach. One could allow for more realistic assumptions, such as dynamic search behavior by pursuers (pursuers update their routes based on new information) and diversity of pursuer types & capability. Edge features could be introduced to account for road cameras, or to incorporate the notion of travel speed and distance between nodes. Our model can accommodate such features, but it would require refinement of the simulation environment and pursuer optimization algorithm.

- Our escape agent model can be used as an evaluation metric to assess the quality of pursuer strategies. In practice, such an application requires the definition of an evaluation protocol that specifies exact training and testing regimes.

- More effort could be made to analyze the behavior of a trained escape agent under different circumstances. Techniques from the emerging field of AI/GNN explainability[YYW+21], or aggregate statistics of escape rollouts may reveal further weaknesses and areas of improvement for pursuer strategies.

Beyond these incremental improvements, the following new directions of work could be worth investigating:

- Our method applies RL to train only the escape agent. An extension to multi-agent reinforcement-learning (MARL[dSNC+21, GEK17, XHG+19]), in which both escaper and pursuers learn under an adversarial training regime might be worth exploring. There is a risk that the behaviors that will emerge are unrealistic, but even if this is the case, it may expose new and relevant escape and/or pursuit patterns. Such an unexpected, novel and previously deemed unproductive pattern (in a different, but related domain) was uncovered by AlphaGo in an adversarial setting [SHMea16, PKKK19].

- There is currently no curated data on real-life behavior of suspects that flee a crime scene. It might be worthwhile to develop such a dataset and potentially distinguish between different types of suspects (less vs. more sophisticated). Data could be gathered from actual pursuits, or from (gamified) simulations. This data could be used to identify escape agent dynamical models, or learning its objectives, values and rewards through inverse reinforcement learning[ACB22].

Chapter 8

Conclusions

The work of his thesis emerged as a continuation of previous work at the Dutch Police on the optimization of pursuer strategy during search & pursuit scenarios when suspects attempt to flee a crime scene. Our research question challenged us to investigate whether and how Reinforcement Learning (RL) can be applied to develop effective escape strategies, as a means to test whether pursuit strategies are vulnerable to exploitation.

Inspired by the emerging field of combinatorial optimization on graphs using RL, a specific Search & Pursuit-Evasion scenario was cast into a graph-based RL environment. In this formulation, a graph state approximates a real road network on which an escaper and an arbitrary amount of pursuit agents have a current position. The escaper has certain target positions, and the pursuit agents act according to some pre-defined strategy (driven by the existing IMA optimizer of the police), assumed to be unknown to the escaper.

We can conclude that it is possible to apply RL and outperform a benchmark heuristic on the escape task. A well designed parametrized policy function can be trained such that it demonstrates sophisticated escape behavior under partial observability. This behavior generalizes to larger and unseen graph topologies and to different initial conditions on these graphs (such as the position of escape and pursuer agents and the set of target nodes for the escaper). Two key components of this model enable such performance and generalization. First, a suitable Graph Neural Network is able to learn to represent graph states in a high-dimensional latent space, and use this representation to select best actions. Graph Attention Networks proved to be best suited for this task. Second, a recurrent neural network, in the form of an LSTM, can be added to the model architecture in order to remember previously encountered graph state representations. This memory state can be used to leverage sporadically obtained information on pursuer positions and further optimize escape behavior in more realistic scenarios with partial observability.

The resulting escape agent model, trained on a particular pursuer strategy, can learn to predict pursuer responses. In other words: it can learn the heuristics required to exploit a particular pursuer strategy. This is a key result because it eliminates the need to develop hand-crafted scenarios to evaluate the exploitability of pursuer strategies. Our results can be used to assess, compare and improve pursuer strategies. Suggested avenues for further improvement include long range planning, allowing for more realistic modeling assumptions and more sophisticated analysis of escape behavior during testing. Multi-agent RL could be applied as a next step to learn escaper and pursuer strategies simultaneously in an adversarial setting.

Bibliography

[ACB22] Stephen Adams, Tyler Cody, and Peter Beling. A survey of inverse reinforcement learning. *Artificial Intelligence Review*, 2022.

[Als04] Brian Alspach. Searching and sweeping graphs: a brief survey. *Le matematiche*, 59(1, 2):5–37, 2004.

[AY21] Uri Alon and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. In *International Conference on Learning Representations (ICLR)*, 2021.

[BAY22] Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks? In *International Conference on Learning Representations (ICLR)*, 2022.

[BBCV21] Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Velickovic. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *ArXiv preprint*, abs/2104.13478, 2021.

[Bea18] Peter Battaglia and Jessica Blake Chandler Hamrick et al. Relational inductive biases, deep learning, and graph networks. *ArXiv preprint*, abs/1806.01261, 2018.

[Ben19] Eric Benhamou. Variance reduction in actor critic methods (ACM). *ArXiv preprint*, abs/1907.09765, 2019.

[BKU15] Ahmet Tunc Bilgin and Esra Kadioglu-Urtis. An approach to multi-agent pursuit evasion games using reinforcement learning. In *International Conference on Advanced Robotics (ICAR)*, pages 164–169. IEEE, 2015.

[BPL+16] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *ArXiv preprint*, abs/1611.09940, 2016.

[BRW+19] Matthew Botvinick, Sam Ritter, Jane X. Wang, Zeb Kurth-Nelson, Charles Blundell, and Demis Hassabis. Reinforcement learning, fast and slow. *Trends in Cognitive Sciences*, 23(5):408–422, 2019.

[CCK+21] Quentin Cappart, Didier Chételat, Elias B. Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2021.

[CHI11] Timothy H Chung, Geoffrey A Hollinger, and Volkan Isler. Search and pursuit-evasion in mobile robotics. *Autonomous robots*, 31(4):299–316, 2011.

[DDS16] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International Conference on Machine Learning (ICML)*, 2016.

[dSNC+21] Cristino de Souza, Rhys Newbury, Akansel Cosgun, Pedro Castillo, Boris Vidolov, and Dana Kulić. Decentralized multi-agent pursuit using deep reinforcement learning. *IEEE Robotics and Automation Letters*, 6(3):4552–4559, 2021.

[FT08] Fedor V. Fomin and Dimitrios M. Thilikos. An annotated bibliography on guaranteed graph searching. *Theoretical Computer Science*, 399(3):236–245, 2008.

[GEK17] Jayesh K Gupta, Maxim Egorov, and Mykel Kochenderfer. Cooperative multi-agent control using deep reinforcement learning. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2017.

[GSR+17] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning (ICML)*, 2017.

[Ham20] William L Hamilton. Graph representation learning. *Synthesis Lectures on Artifical Intelligence and Machine Learning*, 14(3):1–159, 2020.

[HAN+19] Maximilian Hüttenrauch, Sosic Adrian, Gerhard Neumann, et al. Deep reinforcement learning for swarm systems. *Journal of Machine Learning Research*, 20(54):1–31, 2019.

[HL13] Pili Hu and Wing Cheong Lau. A survey and taxonomy of graph sampling. *ArXiv preprint*, abs/1308.5865, 2013.

[HO20] Shengyi Huang and Santiago Ontan'on. A closer look at invalid action masking in policy gradient algorithms. *ArXiv preprint*, abs/2006.14171, 2020.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[HS15] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable MDPs. In *AAAI fall symposium series*, 2015.

[ICfIM65] R. Isaacs, Karreman Mathematics Research Collection, Society for Industrial, and Applied Mathematics. *Differential Games: A Mathematical Theory with Applications to Warfare and Pursuit, Control and Optimization*. SIAM series in applied mathematics. John Wiley Sons, 1965.

[Kas20] Tamar Kastelein. Optimization of intercepting fugitives on the dutch highway. Master's thesis, Vrije Universiteit Amsterdam, School of Business and Economics, 2020.

[KB15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

[KDZ+17] Elias B. Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[KLC98] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1):99–134, 1998.

[KOQ+19] Steven Kapturowski, Georg Ostrovski, John Quan, Rémi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2019.

[KTA19] Boris Knyazev, Graham W. Taylor, and Mohamed R. Amer. Understanding attention and generalization in graph neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[KvHW19] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations (ICLR)*, 2019.

[KW17] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[LWT+17] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *ArXiv preprint*, abs/1907.09765, 2013.

[MMD+19] Sahil Manchanda, Akash Mittal, Anuj Dhawan, Sourav Medya, Sayan Ranu, and Ambuj Singh. Learning heuristics over large graphs via deep reinforcement learning. *ArXiv preprint*, abs/1903.03332, 2019.

[NMTG16] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33, 2016.

[NT20] Chris Nota and Philip S. Thomas. Is the policy gradient a gradient? In *International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, 2020.

[OAMAH15] Shayegan Omidshafiei, Ali-Akbar Agha-Mohammadi, Christopher Amato, and Jonathan P How. Decentralized control of partially observable markov decision processes using belief space macro-actions. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2015.

[Par78] T. D. Parsons. Pursuit-evasion in a graph. In Y. Alavi and Don R. Lick, editors, *Theory and Applications of Graphs*, pages 426–441, Berlin, Heidelberg, 1978. Springer Berlin Heidelberg.

[PCX21] Yun Peng, Byron Choi, and Jianliang Xu. Graph learning for combinatorial optimization: A survey of state-of-the-art. *Data Science and Engineering*, 6, 2021.

[PKKK19] Woohyuk Park, Sungyong Kim, Keunhyoung Luke Kim, and Jeoung Tae Kim. Alphago's decision making. *FLAP*, 6:105–156, 2019.

[PY21] Hyunmok Park and Kijung Yoon. Degree matters: Assessing the generalization of graph neural network. In *IEEE International Conference on Network Intelligence and Digital Content (IC-NIDC)*, 2021.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[SHMea16] David Silver, Aja Huang, and Chris J. Maddison et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[SMea16] John Schulman, Philipp Moritz, and Sergey Levine et al. High-dimensional continuous control using generalized advantage estimation. In *International Conference on Learning Representations (ICLR)*, 2016.

[SO11] Nicholas M. Stiffler and Jason M. O'Kane. Visibility-based pursuit-evasion with probabilistic evader models. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2011.

[SSP+15] Ivan Sorokin, Alexey Seleznev, Mikhail Pavlov, Aleksandr Fedorov, and Anastasiia Ignateva. Deep attention recurrent q-network. *ArXiv preprint*, abs/1512.01693, 2015.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *ArXiv preprint*, abs/1707.06347, 2017.

[SYML18] Doo Re Song, Chuanyu Yang, Christopher McGreavy, and Zhibin Li. Recurrent deterministic policy gradient method for bipedal locomotion on rough terrain challenge. In *IEEE International Conference on Control, Automation, Robotics and Vision (ICARCV)*, 2018.

[VBO+20] Petar Velickovic, Lars Buesing, Matthew C. Overlan, Razvan Pascanu, Oriol Vinyals, and Charles Blundell. Pointer graph networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[VCC+18] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[vD21] I.S. van Droffelaar. Optimizing interception. Research Proposal, Faculty of Technology, Policy, Management, Delft University of Technology, 2021.

[Vel21] Petar Veličković. Theoretical foundations of graph neural networks. In *University of Cambridge CST Wednesday Seminar series*. https://petar-v.com/talks/GNN-Wednesday.pdf, 2021.

[VFJ15] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2015.

[VYP+20] Petar Velickovic, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. In *International Conference on Learning Representations (ICLR)*, 2020.

[Wat89] C.J.C.H Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge, England, 1989.

[WFPS07] Daan Wierstra, Alexander Foerster, Jan Peters, and Juergen Schmidhuber. Solving deep memory pomdps with recurrent policy gradients. In *International conference on artificial neural networks (ICANN)*, 2007.

[WW95] Alan Washburn and Kevin Wood. Two-person zero-sum games for network interdiction. *Operations research*, 43(2):243–251, 1995.

[XHG+19] Lin Xu, Bin Hu, Zhihong Guan, Xinming Cheng, Tao Li, and Jiangwen Xiao. Multi-agent deep reinforcement learning for pursuit-evasion game scalability. In *Chinese Intelligent Systems Conference (CISC)*, 2019.

[XHLJ19] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR)*, 2019.

[XLZ+20] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S. Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? In *International Conference on Learning Representations (ICLR)*, 2020.

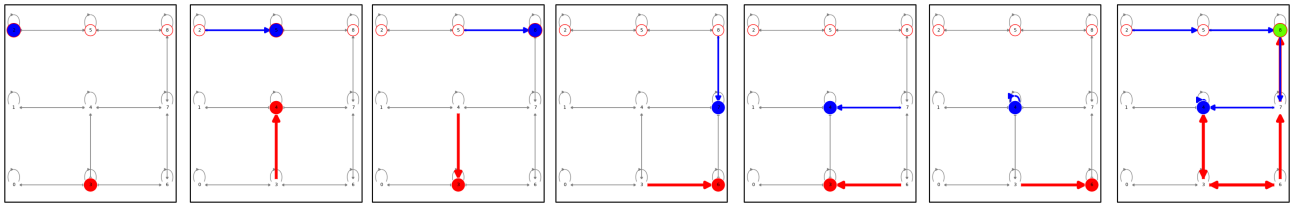
[YWS+19] Yiding Yang, Xinchao Wang, Mingli Song, Junsong Yuan, and Dacheng Tao. SPAGAN: shortest path graph attention network. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2019.

[YYW+21] Hao Yuan, Haiyang Yu, Jie Wang, Kang Li, and Shuiwang Ji. On explainability of graph neural networks via subgraph explorations. In *International Conference on Machine Learning (ICML)*, 2021.

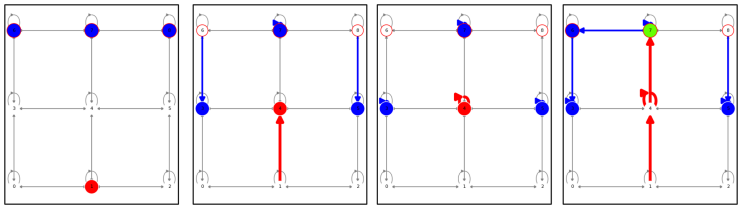
[ZLPM17] Pengfei Zhu, Xin Li, Pascal Poupart, and Guanghui Miao. On improving deep reinforcement learning for pomdps. *ArXiv preprint*, abs/1704.07978, 2017.

Appendix A

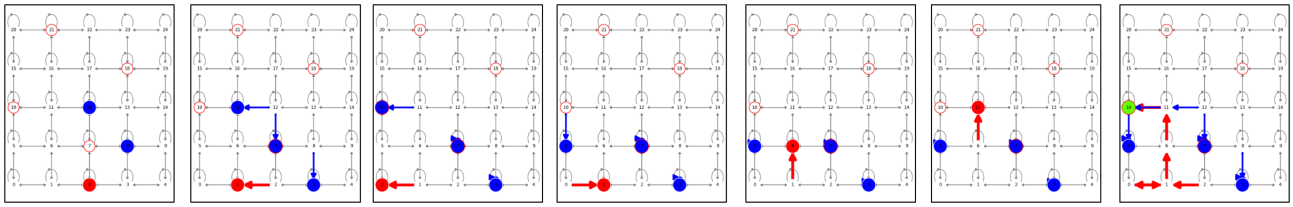
Example graphs & rollouts



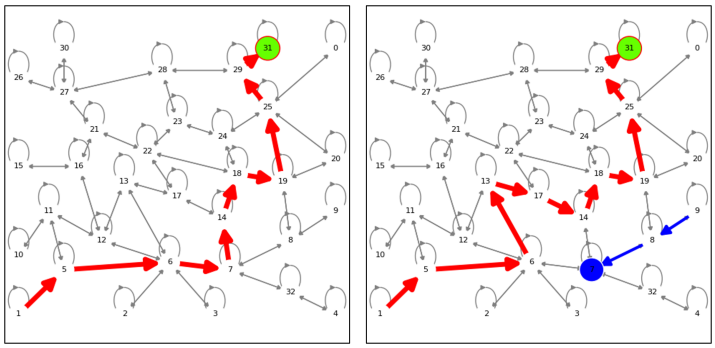
(a) 3x3 Manhattan graph permutation with 3 edges removed, U=1



(b) 3x3 Manhattan graph, optimal solution requires memory (same observation, different actions)

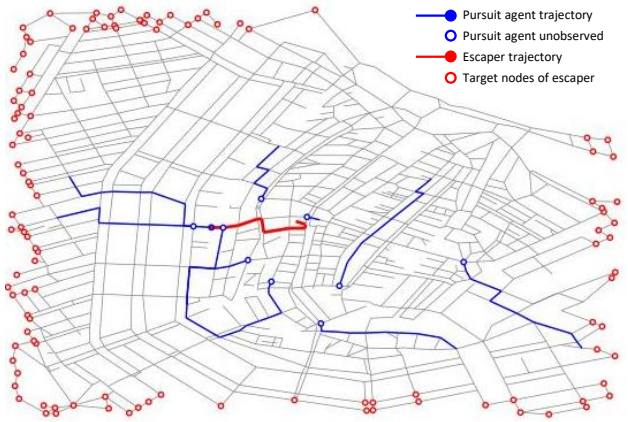


(c) 5x5 Manhattan graph, 4 random target nodes, U=3 static search units

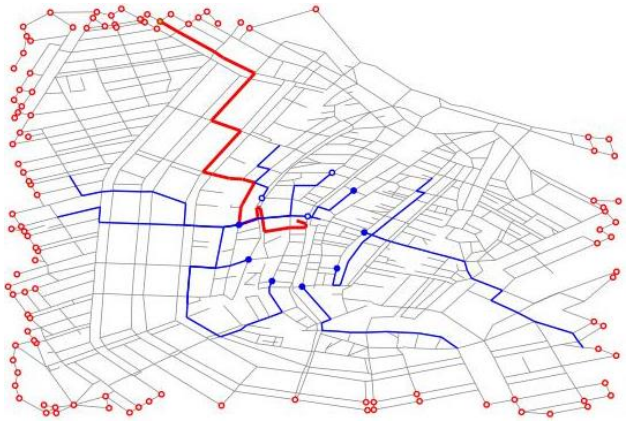


(d) Example Metro graph, N=33 nodes, demonstrating an evasive action, one target node, U=1

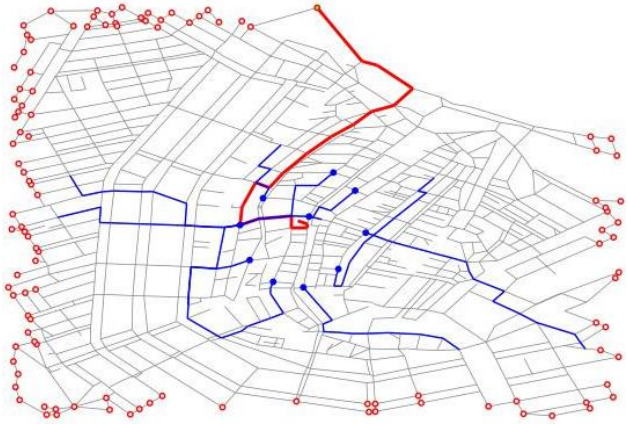
Figure A.1: Illustration of basic graph topologies and rollouts. Passive search agent trajectories (in blue) are provided by a black box optimizer, escape routes (in red) are to be learned



(a) Visibility of 0%, captured after 9 steps



(b) Visibility 50%, 24 steps to target node



(c) Full visibility, 21 steps to target node

Figure A.2: Rollout on an AMS graph instance under different observability levels (baseline GNN-LSTM model, trained on AMS). Final frame is shown.

Appendix B

IMA optimizer for pursuer responses

In this thesis, escape strategies are developed with the aim of exposing weaknesses in pursuer strategies. These pursuers strategies, also referred to as pursuer response functions, were developed in earlier work[vD21] and form the basis of the experimental IMA system (Intelligente Meldkamer Assistent). The optimization is based on Integer Linear Programming (ILP) and formalizes the police-fugitive interception problem as a Flow Interception Problem. The following decision variables are used:

$$\forall v \in V, \forall u \in U, \pi_{u,v} = \begin{cases} 1, & \text{if unit } u \text{ is located at vertex } v \\ 0, & \text{otherwise} \end{cases} \tag{B.1}$$

$$\forall r \in R, z_r = \begin{cases} 1, & \text{if route } r \text{ is intercepted} \\ 0, & \text{otherwise} \end{cases} \tag{B.2}$$

And parameters:

$$\begin{aligned} V &= \{v\} && \text{set of vertices} \\ R &= \{r\} && \text{set of escape routes} \\ U &= \{u\} && \text{set of police units} \\ T &= \{0, 1, \ldots, T_{\max}\} && \text{ordered index set of time steps} \\ \pi_{u,v} &= \{0, 1\} && \text{police unit u is located at vertex v} \\ \phi_{r,v,t} &= \{0, 1\} && \text{escape route r is located at vertex v at time t} \\ \tau_{u,v,t} &= \{0, 1\} && \text{police unit u can reach vertex v at time t} \end{aligned}$$

Given these decision variables and parameters, the optimization is defined as follows:

$$\text{Maximize:} Z^R = \sum_{r \in R} z_r \tag{B.3}$$

$$\text{Subjet to:} \sum_{v \in V} \pi_{u,v} \leq 1 \qquad \forall u \in U \tag{B.4}$$

$$z_r \leq \sum_{u \in U} \sum_{t \in T} \sum_{v \in V} \phi_{r,v,t} \cdot \pi_{u,v} \cdot \tau_{u,t,v} \qquad \forall r \in R \tag{B.5}$$

The basic idea behind this formulation is that there are a number of $|U|$ police units, all with certain initial node positions on the graph. These units are directed to certain interception positions based on knowledge of the start position of the escaper. Escape routes are generated using Monte Carlo simulation, with certain assumptions about the escaper's behavior. The optimizer picks an interception position for each pursuer, such that the fraction of escape routes that are intercepted is maximized. A detailed description of the optimization constraints is offered in [vD21]. The usefulness of this type of orchestration of pursuer units in real-life scenarios has been validated using human-in-the-loop simulations. The IMA optimization serves to assist decision making by operators in operations centers.

Escape route generation Throughout this thesis, two variants of the optimization formulation introduced above are used:

1. In the baseline approach (with pursuer response function denoted as $\mathscr{I}_{base}$), escape routes are generated by assuming the escaper performs non-backtracking random walks. The amount of routes sampled is a hyperparameter ($|R|$) that can be set in the optimization algorithm.

2. In the alternative approach (denoted as $\mathscr{I}_{alt}$), we assume that the pursuers have knowledge about the target nodes that the escaper is trying to reach. For instance, consider a bounding box around the crime scene. A reasonable assumption may be that the escaper is trying to get away from the crime-scene towards any position (a target node) on the edge of this bounding box. The set R in Eqs.B.1 contains all shortest paths from the escaper's initial position towards these target nodes.

These two alternatives lead to very different behaviors of the pursuers. Examples are shown for three city graph topologies in Figures B.1-B.3. Under pursuer response function $\mathscr{I}_{base}$, the pursuit agents tend to spread strategically in an area close to the crime scene (left panels). This can be explained by the fact that as a directed random walker, the escaper is likely to remain in a bounded area around the crime scene. Under $\mathscr{I}_{alt}$, the pursuers tend to spread out more, in their attempt to cover as many routes to target nodes as possible. In our experiments, we demonstrate that $\mathscr{I}_{base}$ constitutes a more effective strategy that is harder to exploit by an escape agent.

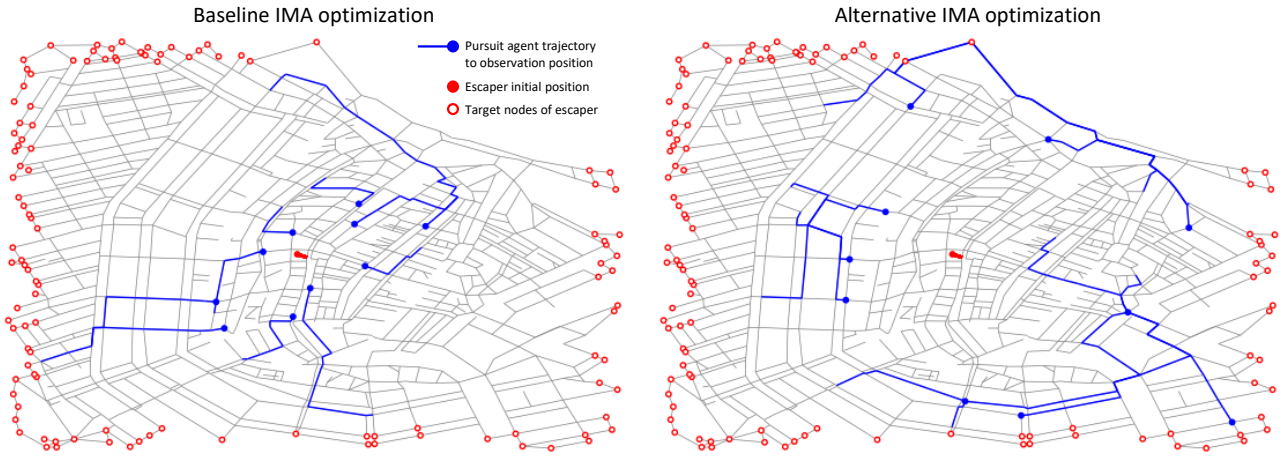


Figure B.1: Pursuer responses on AMS graph, $|U|$=10 with baseline- and alternative IMA optimization

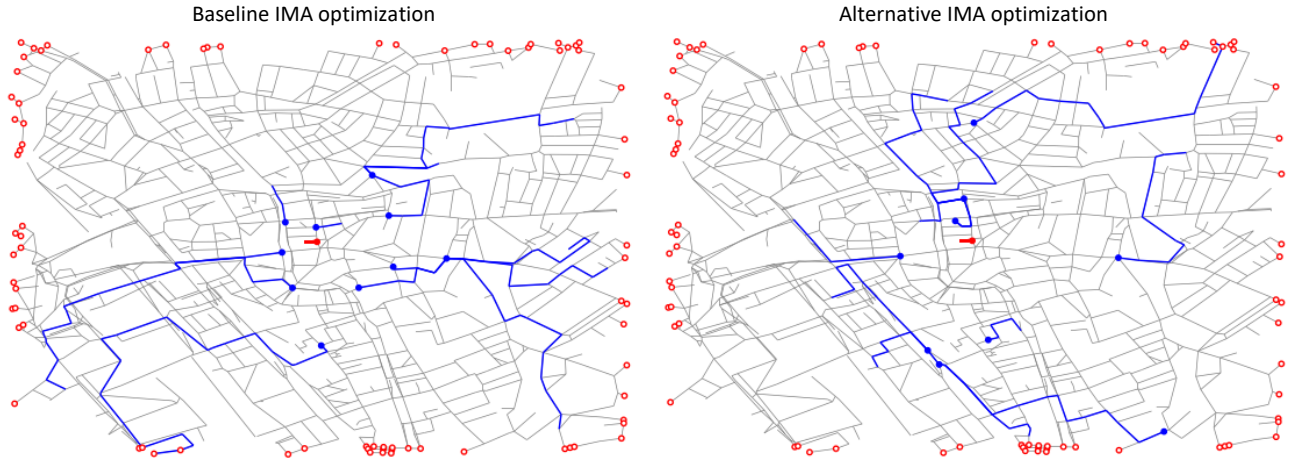


Figure B.2: Pursuer responses on UTR graph, $|U|$=10 with baseline- and alternative IMA optimization

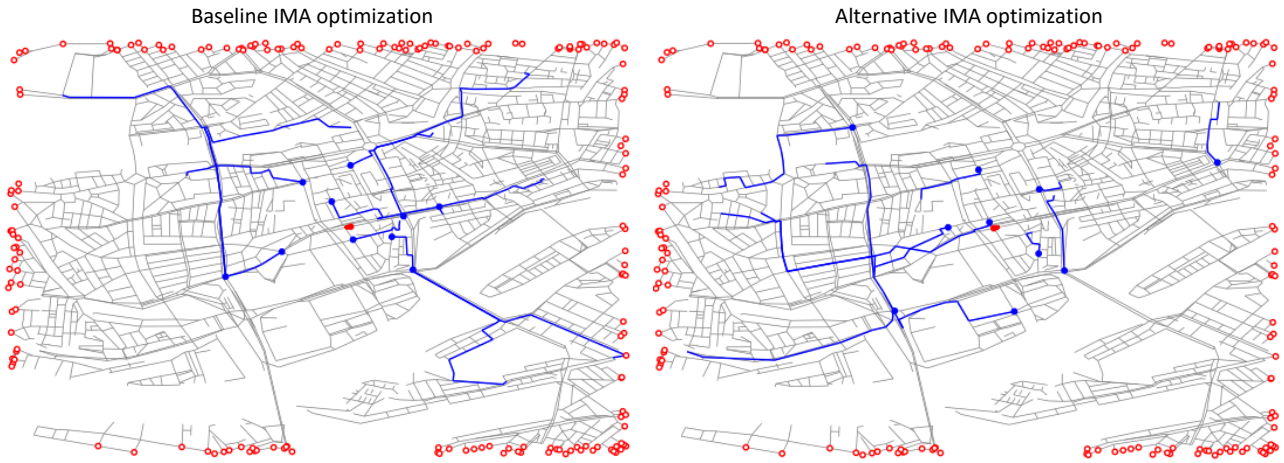


Figure B.3: Pursuer responses on ROT graph, $|U|$=10 with baseline- and alternative IMA optimization

Appendix C

Graph datasets & statistics

The following graph classes have been used in this study to create train- and evaluation environments for the Reinforcement Learning experiments.

Graph classes and topologies

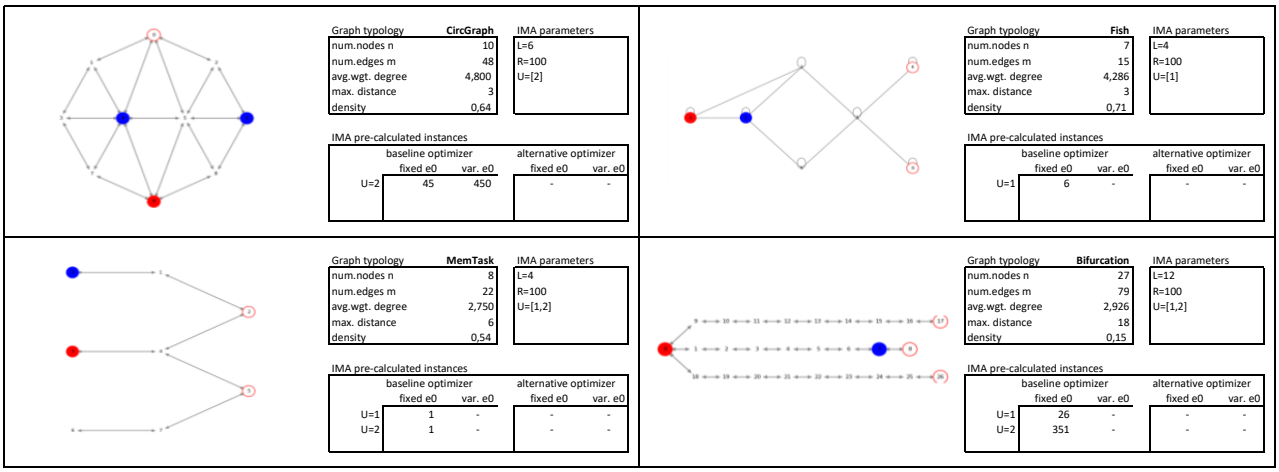


Figure C.1: Demonstration graphs

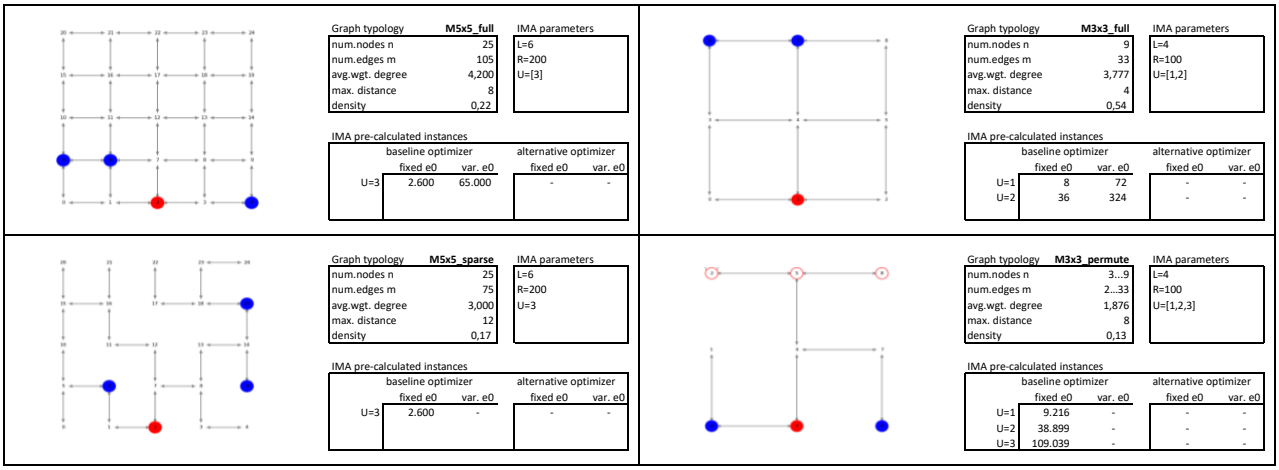
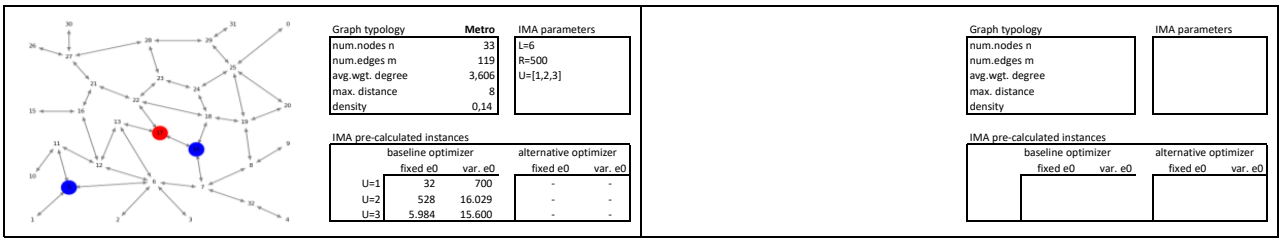


Figure C.2: Manhattan graphs



Graph typology	Metro
num.nodes n	33
num.edges m	119
avg.wgt. degree	3,606
max. distance	8
density	0,14

IMA parameters
L=6
R=500
U=[1,2,3]

IMA pre-calculated instances

	baseline optimizer		alternative optimizer	
	fixed e0	var. e0	fixed e0	var. e0
U=1	32	700	-	-
U=2	528	16.029	-	-
U=3	5.984	15.600	-	-

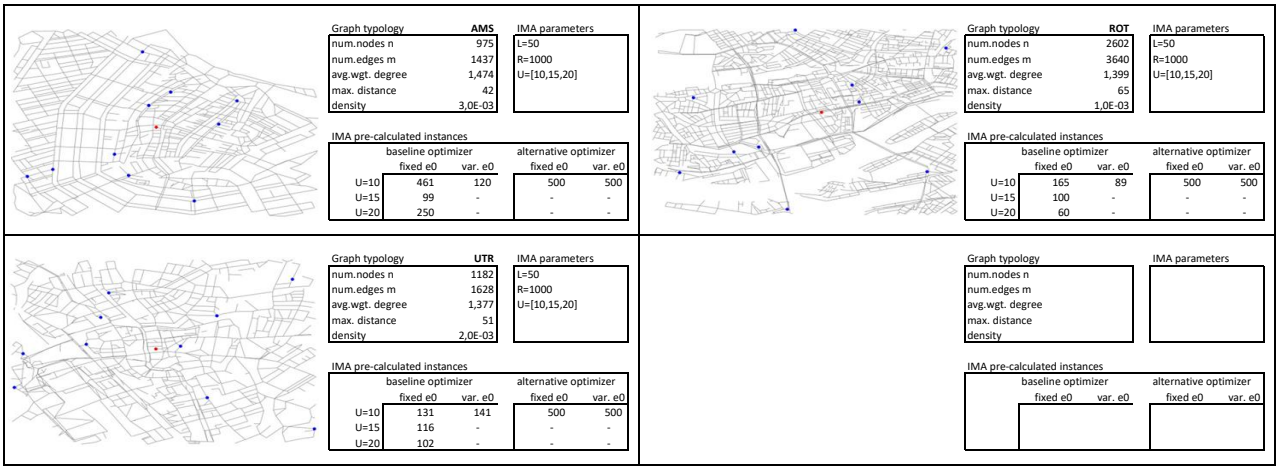
Graph typology
num.nodes n
num.edges m
avg.wgt. degree
max. distance
density

IMA parameters

IMA pre-calculated instances

	baseline optimizer		alternative optimizer	
	fixed e0	var. e0	fixed e0	var. e0

Figure C.3: Highly coarsened city graph



Graph typology	AMS
num.nodes n	975
num.edges m	1437
avg.wgt. degree	1,474
max. distance	42
density	3,0E-03

IMA parameters
L=50
R=1000
U=[10,15,20]

IMA pre-calculated instances

	baseline optimizer		alternative optimizer	
	fixed e0	var. e0	fixed e0	var. e0
U=10	461	120	500	500
U=15	99	-	-	-
U=20	250	-	-	-

Graph typology	ROT
num.nodes n	2602
num.edges m	3640
avg.wgt. degree	1,399
max. distance	65
density	1,0E-03

IMA parameters
L=50
R=1000
U=[10,15,20]

IMA pre-calculated instances

	baseline optimizer		alternative optimizer	
	fixed e0	var. e0	fixed e0	var. e0
U=10	165	89	500	500
U=15	100	-	-	-
U=20	60	-	-	-

Graph typology	UTR
num.nodes n	1182
num.edges m	1628
avg.wgt. degree	1,377
max. distance	51
density	2,0E-03

IMA parameters
L=50
R=1000
U=[10,15,20]

IMA pre-calculated instances

	baseline optimizer		alternative optimizer	
	fixed e0	var. e0	fixed e0	var. e0
U=10	131	141	500	500
U=15	116	-	-	-
U=20	102	-	-	-

Graph typology
num.nodes n
num.edges m
avg.wgt. degree
max. distance
density

IMA parameters

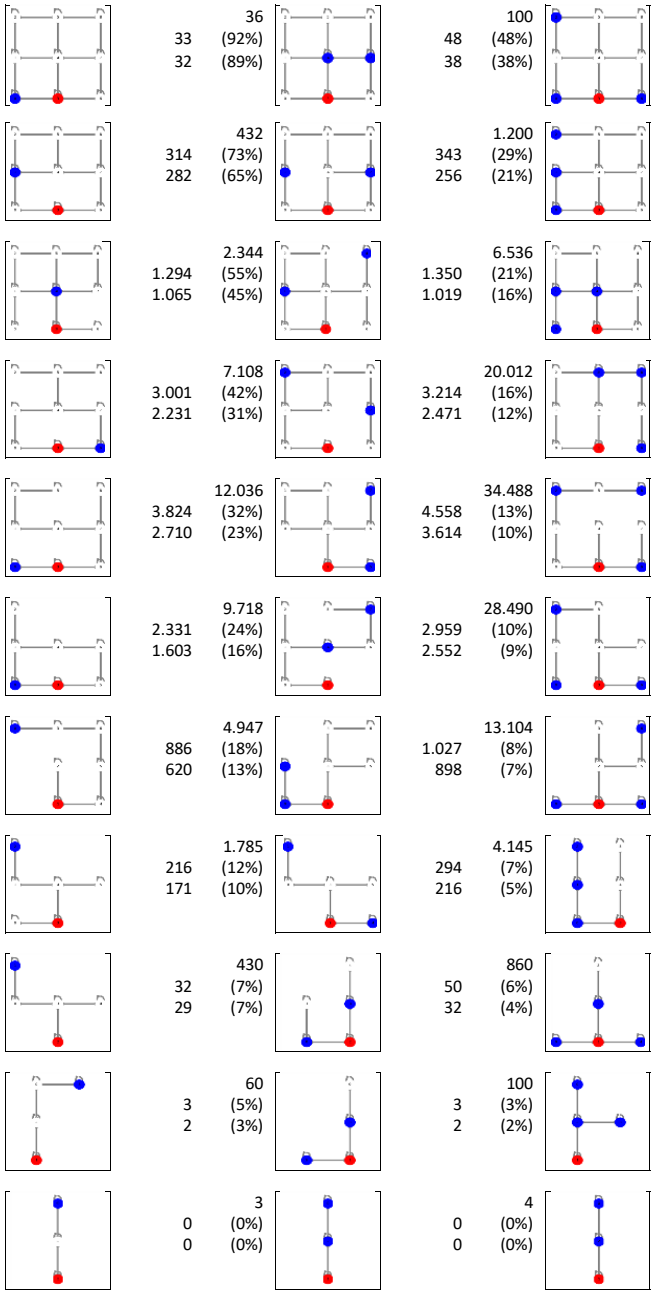
IMA pre-calculated instances

	baseline optimizer		alternative optimizer	
	fixed e0	var. e0	fixed e0	var. e0

Figure C.4: Large scale city graphs

Manhattan 3x3 graph permutations

Edges removed*	Eligible graphs**		Number of pursuers: 1		2		3	
0	1	Total valid instances	8		36		100	
		Solvable, w/o edge captures	8	(100%)	33	(92%)	48	(48%)
		with edge captures	8	(100%)	32	(89%)	38	(38%)
1	12	Total valid instances	96		432		1.200	
		Solvable, w/o edge captures	91	(95%)	314	(73%)	343	(29%)
		with edge captures	91	(95%)	282	(65%)	256	(21%)
2	66	Total valid instances	524		2.344		6.536	
		Solvable, w/o edge captures	454	(87%)	1.294	(55%)	1.350	(21%)
		with edge captures	435	(83%)	1.065	(45%)	1.019	(16%)
3	207	Total valid instances	1.613		7.108		20.012	
		Solvable, w/o edge captures	1.233	(76%)	3.001	(42%)	3.214	(16%)
		with edge captures	1.083	(67%)	2.231	(31%)	2.471	(12%)
4	378	Total valid instances	2.826		12.036		34.488	
		Solvable, w/o edge captures	1.895	(67%)	3.824	(32%)	4.558	(13%)
		with edge captures	1.422	(50%)	2.710	(23%)	3.614	(10%)
5	371	Total valid instances	2.501		9.718		28.490	
		Solvable, w/o edge captures	1.476	(59%)	2.331	(24%)	2.959	(10%)
		with edge captures	981	(39%)	1.603	(16%)	2.552	(9%)
6	246	Total valid instances	1.444		4.947		13.104	
		Solvable, w/o edge captures	763	(53%)	886	(18%)	1.027	(8%)
		with edge captures	444	(31%)	620	(13%)	898	(7%)
7	121	Total valid instances	599		1.785		4.145	
		Solvable, w/o edge captures	289	(48%)	216	(12%)	294	(7%)
		with edge captures	136	(23%)	171	(10%)	216	(5%)
8	43	Total valid instances	172		430		860	
		Solvable, w/o edge captures	65	(38%)	32	(7%)	50	(6%)
		with edge captures	26	(15%)	29	(7%)	32	(4%)
9	10	Total valid instances	30		60		100	
		Solvable, w/o edge captures	8	(27%)	3	(5%)	3	(3%)
		with edge captures	2	(7%)	2	(3%)	2	(2%)
10	1	Total valid instances	2		3		4	
		Solvable, w/o edge captures	0	(0%)	0	(0%)	0	(0%)
		with edge captures	0	(0%)	0	(0%)	0	(0%)



* While reachability of at least one target node (the top three nodes in each graph) by the escape agent is maintained

** Only one connected component that includes the escape agent's initial position is accepted. Isomorphic sub-graphs are kept in the sets as we assume no knowledge of how the pursuit unit positions are optimized (this may or may depend on geographic positions of the nodes)

● = Escape agent ● = Pursuit unit

Figure C.5: Manhattan 3x3 graph permutations

Appendix D

Implementation details

Node feature definitions

Throughout this study, different node feature definitions are used. We write a node feature for node v at time t as $\boldsymbol{x}_{v,t} = \left(i_{v,t}^{(\cdot)}, ..., i_{v,t}^{(\cdot)}\right)^T \in \mathbb{R}^f$, where $i^{(\cdot)}$ denotes an element of the node feature that can be chosen from a range of elements that we define below. The information contained in the node features is key to the ability of the GNN-based models to learn useful latent node representations. An illustration of node features is shown in Fig.D.1 and the nomenclature of the different feature definitions used in this study is provided in Fig.D.2. The feature elements are defined next.

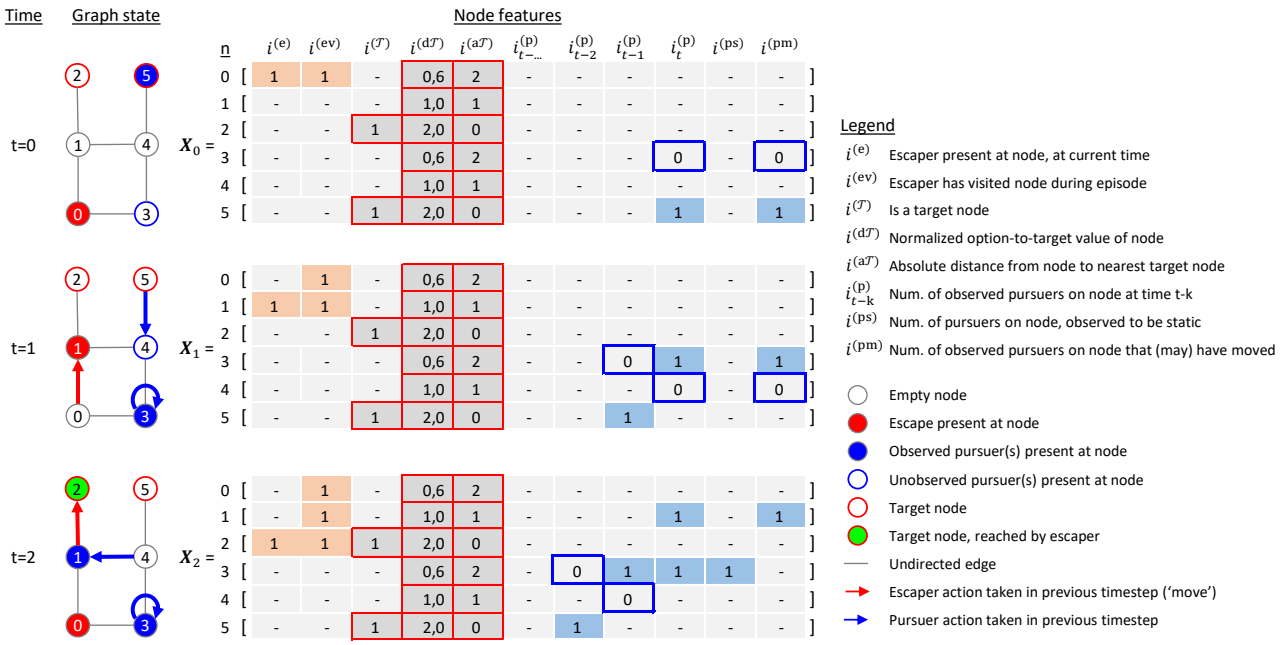


Figure D.1: Illustration of node feature values under partial observability for a basic 6-node graph instance (U=2 pursuers, target node set $\mathcal{T} = \{2, 5\}$, initial positions $e_0 = (0)$, $\boldsymbol{u}_0 = (3, 5)$, $p_{vis} = 0.5$)

Node feature nomenclature	$i^{(e)}$	$i^{(ev)}$	$i^{(\mathcal{T})}$	$i^{(d\mathcal{T})}$	$i^{(a\mathcal{T})}$	$i_{t-...}^{(p)}$	$i_{t-2}^{(p)}$	$i_{t-1}^{(p)}$	$i_{t}^{(p)}$	$i^{(ps)}$	$i^{(pm)}$	Dim	Nomenclature used in the code
NFM_3	✓		✓						✓			3	NFM_ec_t_u
NFM_4	✓	✓	✓						✓			4	NFM_ev_ec_t_u
NFM_5	✓	✓	✓							✓	✓	5	NFM_ev_ec_t_um_us
NFM_7	✓	✓	✓	✓	✓					✓	✓	7	NFM_ev_ec_t_dt_at_um_us
NFM_10	✓	✓	✓	✓	✓	✓	✓	✓	✓			10	NFM_ev_ec_t_dt_at_ustack5

Figure D.2: Node feature nomenclature as used in the experiments

Node feature definitions Our main approach to node feature definitions has been to keep them as basic as possible and encode the minimum information content required to learn and leverage an escape strategy. However, in order to achieve sufficient baseline performance some level of feature engineering proved to be necessary. Two main obstacles needed to be dealt with:

1. Pathfinding to long-distance target nodes requires that target node information is visible within the GNN's perceptive field for a given node. We typically use 5 layers in the GNN architectures which means node features can only contain information from other nodes that are at most 5 hops away. This is problematic for larger graphs where the distance to target nodes be much larger. We solve this by augmenting the node features with two elements:
   (a) a normalized 'option value' (a high option value means many target nodes are reachable and nearby), which provides a raw signal indicating attractive nodes that will lead the escaper to profitable paths if no pursuers were present;
   (b) the absolute distance to the nearest target node, which provides a signal to assist value estimation in Critic- or Q-networks.
2. Under partial observability scenarios, a form of memory of past actions and past observations is required. One way to achieve this is to apply 'frame-stacking', which means node features are extended with selected information from observations made in previous timesteps.

This leads to the following definition of the (optional) feature elements of a node v at timestep t:

- $i_{v,t}^{(\mathrm{e})} \in \{0, 1\}$, a Boolean indicator: escaper is located on node v at current time t

$$i_{v,t}^{(\mathrm{e})} = \begin{cases} 1, \text{ if } e_t = v \\ 0, \text{ otherwise} \end{cases} \tag{D.1}$$

- $i_{v,t}^{(\mathrm{ev})} \in \{0, 1\}$, a Boolean indicator: node v has been visited by the escaper in previous timesteps

$$i_{v,t}^{(\mathrm{ev})} = \begin{cases} 1, \text{ if } \exists t' \leq t \text{ s.t. } e_{t'} = v \\ 0, \text{ otherwise} \end{cases} \tag{D.2}$$

- $i_v^{(\mathcal{T})} \in \{0, 1\}$, a Boolean indicator: node v is a target node that the escaper is aiming to reach

$$i_v^{(\mathcal{T})} = \begin{cases} 1, \text{ if } v \in \mathcal{T} \\ 0, \text{ otherwise} \end{cases} \tag{D.3}$$

  Note that this indicator does not depend on time: target nodes are assumed to be constant on a given graph instance.

- $i_v^{(d\mathcal{T})} \in [0, 2]$, the normalized option-to-target value of node v, defined as:

$$i_v^{(d\mathcal{T})} = \begin{cases} 0, \text{ if there is no directed path from node } v \text{ to any target node } w \in \mathcal{T} \\ 2, \text{ if node } v \text{ is a target node, } v \in \mathcal{T} \\ \eta \sum_{w \in \mathcal{T}} \frac{1}{d(v,w)}, \text{ otherwise} \end{cases} \tag{D.4}$$

  where $\mathcal{T}$ is the target node set, $d(v, w)$ the shortest path distance from node v to node w, η is a normalizing constant such that $\max_{u \notin \mathcal{T}} \left(i_u^{(d\mathcal{T})}\right) = 1$. Normalizing this indicator ensures that it provides a useful signal for any arbitrary graph size. Note that this indicator does not depend on time: graph structure and target nodes are assumed to be constant on a given graph instance.

- $i_v^{(a\mathcal{T})} \in \mathbb{Z}_{\geq 0}$, the distance from node v to the nearest target node $w \in \mathcal{T}$

$$i_v^{(a\mathcal{T})} = \min_{w \in \mathcal{T}} d(v, w) \tag{D.5}$$

- $i^{(p)}_{v,t-k} \in \mathbb{Z}_{\geq 0}$, the number of observed[1] pursuers on node v at timestep $t-k$, with $k \geq 0$

$$i^{(p)}_{v,t-k} = \sum_{j=1}^{U} m^{(j)}_{t-k} \cdot \mathbb{1}_{u^{(j)}_{t-k}=v} \tag{D.6}$$

  This calculation involves the visibility mask value of the j-th pursuer $m^{(j)}_t \sim Bernoulli(p_{vis})$, where $p_{vis} \in [0,1]$ is a visibility probability that can be chosen to define partial observability of the environment. Pursuer positions are denoted $\boldsymbol{u}_t = \left(u^{(1)}_t, ..., u^{(U)}_t\right)$ and $\mathbb{1}$ is the indicator function. We refer to the use node feature elements $i^{(p)}_{v,t-k}, ..., i^{(p)}_{v,t}$ with $k > 0$ as 'frame-stacking'; past observations of pursuit agents are stacked in the node features of the graph.

- $i^{(ps)}_{v,t} \in \mathbb{Z}_{\geq 0}$, the number of observed pursuers on node u that have not moved in the last timestep

$$i^{(ps)}_{v,t} = \sum_{j=1}^{U} m^{(j)}_{t-1} \cdot m^{(j)}_t \cdot \mathbb{1}_{u^{(j)}_{t-1}=v} \cdot \mathbb{1}_{u^{(j)}_t=v} \tag{D.7}$$

- $i^{(pm)}_{v,t} \in \mathbb{Z}_{\geq 0}$, the number of observed pursuers on node u that (may) have moved in the last timestep

$$i^{(pm)}_{v,t} = i^{(p)}_{v,t} - i^{(ps)}_{v,t} \tag{D.8}$$

Although these definitions may seem daunting, the application of the various node feature elements is rather intuitive, Figure D.1 aims to illustrate this with an example.

Code used for experiments

Repository with codebase used for all experiments: `https://github.com/rvdweerd/simmodel`

Environment class

Gym environment:`https://github.com/rvdweerd/simmodel/blob/5d074f6f75082b57f0a3340162abd99576f92ae4/modules/rl/environments.py#L21`

NFM classes: `https://github.com/rvdweerd/simmodel/blob/5d074f6f75082b57f0a3340162abd99576f92ae4/modules/gnn/nfm_gen.py#L425`

Gym wrappers: `https://github.com/rvdweerd/simmodel/blob/5d074f6f75082b57f0a3340162abd99576f92ae4/modules/ppo/ppo_wrappers.py#L251`

IMA optimizer code

`https://github.com/rvdweerd/simmodel/blob/5d074f6f75082b57f0a3340162abd99576f92ae4/modules/optim/simdata_create.py#L19`

CRA heuristic policy

Risk esimator: `https://github.com/rvdweerd/simmodel/blob/5d074f6f75082b57f0a3340162abd99576f92ae4/modules/rl/rl_policy.py#L210`

Policy class: `https://github.com/rvdweerd/simmodel/blob/5d074f6f75082b57f0a3340162abd99576f92ae4/modules/ppo/ppo_wrappers.py#L420`

DQN, own implementation

Model class: `https://github.com/rvdweerd/simmodel/blob/5d074f6f75082b57f0a3340162abd99576f92ae4/modules/gnn/comb_opt.py#L442`

Train function: `https://github.com/rvdweerd/simmodel/blob/5d074f6f75082b57f0a3340162abd99576f92ae4/Phase2b_gnn-dqn.py#L95`

PPO, SB3 implementation

Model class: `https://github.com/rvdweerd/simmodel/blob/5d074f6f75082b57f0a3340162abd99576f92ae4/modules/ppo/models_sb3_gat2.py#L240`

Train function: `https://github.com/rvdweerd/simmodel/blob/5d074f6f75082b57f0a3340162abd99576f92ae4/Phase2c_gnn-ppo_sb3.py#L75`

PPO-GNN-LSTM, own implementation

Model class: `https://github.com/rvdweerd/simmodel/blob/5d074f6f75082b57f0a3340162abd99576f92ae4/modules/ppo/models_basic_lstm.py#L408`

Train function: `https://github.com/rvdweerd/simmodel/blob/5d074f6f75082b57f0a3340162abd99576f92ae4/Phase3_lstm-gnn-ppo_simp.py#L44`

[1] Observed here means observed by the escaper

Train- and testset definitions

Trainset	Graph class	Nodes	U	Escape initial position	Target nodes	# topologies	# instances	Sample weight
M3M5								
U01234/E012345678[1]		5-9	[0-4]	Fixed (bottom center)	Fixed (3 top nodes)	1.456	6k	0,38
	M5x5_full.V	25	3	Variable	Fixed (5 top nodes)	1	64k	0,15
	M5x5_full.V	25	3	Variable	Variable, ~[1-3]	1	1E+09	0,15
	M5x5_full.F	25	3	Fixed (bottom center)	Fixed (5 top nodes)	1	3k	0,08
	M5x5_full.F	25	3	Fixed (bottom center)	Variable, ~[1-3]	1	8E+07	0,08
	M5x5_full.F	25	0	Fixed (bottom center)	Variable, ~[1]	1	25	0,08
	M5x5_sparse.F	25	3	Fixed (bottom center)	Variable, ~[1-3]	1	8E+07	0,08
								1,00
M*								
	Metro.V	33	3	Variable	Fixed (12 border nodes)	1	16k	0,05
	Metro.V	33	3	Variable	Variable, ~[1-2]	1	2E+07	0,05
	Metro.F	33	3	Fixed (center)	Fixed (12 border nodes)	1	6k	0,05
	Metro.F	33	3	Fixed (center)	Variable, ~[1-2]	1	7E+06	0,05
	Metro.F	33	3	Fixed (bottom left)	Fixed (1 top right)	1	6k	0,05
	Metro.F	33	3	Fixed (bottom left)	Variable, ~[1-2]	1	7E+06	0,05
U01234/E012345678[1]		5-9	[0-4]	Fixed (bottom center)	Fixed (3 top nodes)	1.456	6k	0,26
	M5x5_full.V	25	3	Variable	Fixed (5 top nodes)	1	64k	0,11
	M5x5_full.V	25	3	Variable	Variable, ~[1-3]	1	1E+09	0,11
	M5x5_full.F	25	3	Fixed (bottom center)	Fixed (5 top nodes)	1	3k	0,05
	M5x5_full.F	25	3	Fixed (bottom center)	Variable, ~[1-3]	1	8E+07	0,05
	M5x5_full.F	25	0	Fixed (bottom center)	Variable, ~[1]	1	25	0,05
	M5x5_sparse.F	25	3	Fixed (bottom center)	Variable, ~[1-3]	1	8E+07	0,05
								1,00
AMS								
	AMS.V	975	10	Variable	Fixed (113 border nodes)	1	120	0,25
	AMS.V	975	10	Variable	Variable, ~[10-20]	1	>>1E+09	0,25
	AMS.F	975	10	Fixed (center)	Fixed (113 border nodes)	1	461	0,25
	AMS.F	975	10	Fixed (center)	Variable, ~[10-20]	1	>>1E+09	0,25
								1,00

1) E0-8 indicates all Manhattan 3x3 graph permutations created by removing 0-8 edges are included in the trainset

Figure D.3: Definition of trainsets used in the experiments

Testset	Graph class	Nodes	U	Escape initial position	Target nodes	# topologies	# instances	Sample weight
M3								
U01234/E012345678[1]		5-9	[0-4]	Fixed (bottom center)	Fixed (3 top nodes)	1.456	6k	N/a[2]
M5								
	M5x5_full.V	25	3	Variable	Fixed (5 top nodes)	1	64k	N/a[2]
	M5x5_full.F	25	3	Fixed (bottom center)	Fixed (5 top nodes)	1	3k	N/a[2]
Metro								
	Metro.V	33	3	Variable	Fixed (12 border nodes)	1	16k	N/a[2]
	Metro.F	33	3	Fixed (center)	Fixed (12 border nodes)	1	6k	N/a[2]
AMS								
	AMS.V	975	10	Variable	Fixed (113 border nodes)	1	120	N/a[2]
	AMS.F	975	10	Fixed (center)	Fixed (113 border nodes)	1	461	N/a[2]
UTR								
	UTR.V	1182	10	Variable	Fixed (79 border nodes)	1	141	N/a[2]
	UTR.F	1182	10	Fixed (center)	Fixed (79 border nodes)	1	131	N/a[2]
ROT								
	ROT.V	2602	10	Variable	Fixed (173 border nodes)	1	89	N/a[2]
	ROT.F	2602	10	Fixed (center)	Fixed (173 border nodes)	1	165	N/a[2]

1) E0-8 indicates all Manhattan 3x3 graph permutations created by removing 0-8 edges are included in the trainset
2) During testing, all graph instances are inclued (no sampling is used)

Figure D.4: Definition of testsets used in the experiments

Model architectures and hyperparameters

		Experiment 6.1-6.2							Experiment 6.3-6.5		LSTM Experiments				
Algoritm		DQN			PPO				PPO		PPO				
GNN Feature extractor		GATv2	s2v	s2v	GATv2	GATv2	GATv2	s2v	GATv2	GATv2	GATv2	GATv2	GATv2	GATv2	GATv2
Designation code		A	B	C	A	B	C	D	LSTM.E	F	LSTM.G	LSTM.H	LSTM.I	LSTM.J	K
Implementation		Own	Own	Own	SB3	Own-RS[1]	Own-RS[1]	SB3	Own-FS[1]	Own-FS[1]	Own-FS[1]	Own-FS[1]	Own-FS[1]	Own-FS[1]	Own-FS[1]
Agent observation space															
Node feature def		NFM_7[2]	NFM_4[2]	NFM_7[2]	NFM_7[2]	NFM_7[2]	NFM_7[2]	NFM_7[2]	NFM_7[2]	NFM_10[2]	NFM_7[2]	NFM_7[2]	NFM_7[2]	NFM_7[2]	NFM_8[2]
Node feature dim		7	4	7	7	7	7	7	7	10	7	7	7	7	8
Observability		full	full	full	full	full	full	full	partial[3]	partial[3]	custom[4]	custom[4]	custom[4]	custom[4]	custom[4]
Model architecture															
GAT	heads	2	-	-	1	2	1	-	4	4	4	4	4	4	4
	layers	5	-	-	5	5	5	-	5	5	5	5	5	5	5
	concat	True	-	-	True	True	True	-	True	True	True	True	True	True	True
	share weights	False	-	-	False	True	False	-	False	False	False	False	False	False	False
	hidden channels	64	-	-	64	64	64	-	128	128	48	48	48	48	48
	out channels	64			64	64	64	-	64	64	24	24	24	24	24
s2v	propag.steps	-	3	5	-	-	-	5	-	-	-	-	-	-	-
	latent dim	-	64	64	-		-	64	-	-	-	-	-	-	-
	theta1 layers	-	2	2	-		-	2	-	-	-	-	-	-	-
LSTM	Placement	None	None	None	None	None	None	None	EMB	None	EMB	FE	Dual	DualCC	None
	Hidden dim	-	-	-	-	-	-	-	64	-	24	24	24	24	-
	Num layers	-	-	-	-	-	-	-	1	-	1	1	1	1	-
Actor	Latent dim	-	-	-	64	64	64	64	64	64	24	24	24	24	24
Critic	variant	-	-	-	MO[5]	MO[5]	MO[5]	MO[5]	MO[5]	MO[5]	MO+LIN[5]	MO+LIN[5]	MO+LIN[5]	MO+LIN[5]	MO+LIN[5]
	Latent dim	-	-	-	64	64	64	64	64	64	24	24	24	24	24
Qnet	latent dim	-	64	64	-	-	-	-	-	-	-	-	-	-	-
	#Parameters	43.393	21.569	21.569	34.944	34.690	51.842	30.022	168.962	136.450	24.962	20.610	29.762	57.794	20.450
Hyperparameters General															
	Optimizer	Adam	Adam	Adam	Adam	Adam	Adam	Adam	Adam	Adam	Adam	Adam	Adam	Adam	Adam
	Batch size	48	48	48	64	48	48	64	Variable[6]	Variable[6]	Variable[6]	Variable[6]	Variable[6]	Variable[6]	Variable[6]
	discount factor	0,95	0,95	0,95	0,99	0,98	0,98	0,99	0,98	0,98	0,98	0,98	0,98	0,98	0,98
	learning rate	1,0E-03	1,0E-03	1,0E-03	3E-04	5E-04	5E-04	3E-04	5E-04	5E-04	5E-04	5E-04	5E-04	5E-04	5E-04
Hyperparameters DQN															
	Replay mem size	5.000	2.000	5.000	-	-	-	-	-	-	-	-	-	-	-
	Target update tau	100	100	100	-	-	-	-	-	-	-	-	-	-	-
Hyperparameters PPO															
	entropy loss coef	-	-	-	0	0	0	0	0	0	0	0	0	0	0
	value loss coeff	-	-	-	0,5	1,0	1,0	0,5	1,0	1,0	1,0	1,0	1,0	1,0	1,0
	GAE lambda	-	-	-	0,95	0,95	0,95	0,95	0,95	0,95	0,95	0,95	0,95	0,95	0,95
	Max grad norm	-	-	-	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5
	PPO-clip	-	-	-	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2
	PPO-epochs	-	-	-	10	10	10	10	2	2	2	2	2	2	2
	Rollout steps	-	-	-	2048	100	100	2048	100	100	100	100	100	100	100
	Full episode rollouts	-	-	-	N/a	N/a	N/a	N/a	4	4	4	4	4	4	4

1) Own-RS: own implementation using random sampling of sequences; Own-FS: using full rollouts for gradient updates
2) Node feature definitions: NFM_4=NFM_ev_ec_t_u, NFM_7=NFM_ev_ec_t_dt_at_um_us, NFM_10=NFM_ev_ec_t_dt_at_stack5, NFM_8=NFM_ev_ec_t_dt_at_stack3
3) During training, p_vis ~U({1.0,0.9,0.8,0.7,0.6,0.5})
4) During training on MemTask, pursuer position are only visible in the first frame
5) MO = Max operator variant of Critic, LIN = Linear mapping variant of Critic
6) Batch size equals number of sequences generated in episode rollout stage

Figure D.5: Definition of model architectures and trainsets used in the experiments

Appendix E

Additional experiments

1. Scale-up experiment, detailed results

Variation of train- and testset composition We are interested in the performance of models trained using sub-sets of small graphs, evaluated on sub-sets of larger (unseen) graphs. For this purpose, we expand the diversity of training sets and test sets to form the evaluation matrix shown in Fig.E.1.

Along the rows of this matrix, we vary the RL algorithm used (DQN or PPO) and the GNN used to extract latent node features (*GATv2* or *struc2vec*). We limit model complexity and keep their sizes under 30k parameters for now (details on model definitions and hyperparameters are provided in Appendix D). Each of these variants are trained under full observability, using a different trainset, ranging from small graphs (denoted M3M5) to a mix of small and large graphs (AMS+M*)D. Along the columns of this matrix, the performance on different testsets is shown, again ranging from testsets with small graphs (denoted M3) to large graphs (AMS, UTR, ROT). Like before, a red border around a cell in this matrix indicates that the testset is a subset of the trainset. Color coding is used to indicate performance on unseen graphs (testsets that are not subsets of trainsets), relative to the CRA Heuristic. At this stage, we are only interested in a proof of concept of our approach: can our models outperform the CRA heuristic at all. To test this, the success rates shown in Fig.E.1 are based on the best of 5 seeds. Several observations can be made (references are to the line numbers in Fig.E.1):

Performance of model variants on train- and testsets (% Success Rate)

Tested on (best of 5 seeds / standard deviation)

	Variant	Trainset	M3	M5	Metro	AMS	UTR	ROT
1	dqn.gat	M3M5	94% 2%	94% 3%	94% 7%	78% 17%	66% 11%	88% 18%
2	dqn.gat	M*	92% 1%	93% 3%	99% 1%	67% 14%	61% 12%	84% 22%
3	dqn.gat	AMS	57% 8%	78% 11%	94% 8%	95% 1%	95% 3%	96% 1%
4	dqn.gat	AMS+M*	90% 2%	94% 5%	96% 3%	93% 2%	89% 4%	95% 2%
5	dqn.s2v	M3M5	94% 0%	96% 1%	98% 2%	4% 1%	5% 1%	4% 1%
6	dqn.s2v	M*	95% 14%	96% 1%	99% 0%	4% 1%	5% 2%	6% 2%
7	dqn.s2v	AMS	43% 11%	30% 13%	31% 12%	93% 2%	85% 2%	77% 24%
			M3 std	M5 std	Metro std	AMS std	UTR std	ROT std
8	ppo.gat	M3M5	90% 3%	94% 18%	95% 25%	71% 9%	53% 3%	89% 7%
9	ppo.gat	M*	90% 7%	63% 6%	97% 9%	67% 17%	51% 14%	87% 27%
10	ppo.gat	AMS	79% 10%	69% 14%	94% 9%	95% 6%	94% 11%	96% 2%
11	ppo.gat	AMS+M*	87% 4%	88% 16%	98% 5%	92% 7%	86% 9%	82% 11%
12	PPO.s2v	M3M5	95% 0%	97% 4%	81% 18%	23% 9%	21% 7%	22% 6%
13	PPO.s2v	M*	96% 1%	92% 6%	98% 2%	44% 15%	39% 14%	27% 9%
14	PPO.s2v	AMS	86% 8%	63% 7%	66% 1%	62% 1%	53% 2%	80% 2%
	Sh.path heuristic		64%	48%	62%	49%	46%	73%
	CRA heuristic		95%	84%	93%	88%	79%	91%

Performance on trainset

Performance on testset relative to CRA Heuristic

Figure E.1: Results, scalability experiment

- Lines 1,8 show that learning on small graphs can transfer to larger graphs. The shortest path heuristic can be outperformed, but this is not the case for the (more sophisticated) CRA heuristic.
- Lines 3,10 show that learning solely on instances of the AMS graph topology transfers to other large graph topologies (UTR, ROT). It also transfers to smaller topologies (Metro, M5) although it fails on the smallest and very local evasion tasks that are part of the M3 testset.
- Lines 4,11 shows that expanding the range of experience during training enables the models to improve performance across the entire range of testsets.

- Lines 5-7, 13-14 show that the *struc2vec*-based GNNs cannot match *GATv2*'s performance. The *struc2vec* expressiveness cannot be expanded straightforwardly, contrary to *GATv2* (here, only two attention heads are used and weights are shared between *GATv2* layers).
- The variance over different seeds of test performance tends to be lower for DQN (lines 1-7) vs. PPO (lines 8-14).

Effect of increasing the number of GNN layers Effective navigation by an escape agent over longer graph distances requires the ability to sense distant node feature information. For instance, if the escape agent is located on some node v on a graph, and there is a target node w that can be reached in $k = 10$ steps, the node feature entry that encodes target node status has to 'reach' the escape agent. If a GNN is used to represent the graph state, the propagation of that information over 10 hops will require $k - 1 = 9$ GNN layers to be stacked, assuming the escape agent model's value or policy function (see Eq.5.8 and 5.9) is based on the embeddings of v's neighbors $\mathcal{N}_v$. Long distance information propagation by GNNs can be problematic. It has been shown [AY21] that stacking too many GNN layers in order to increase the receptive field can lead to information bottlenecks. This effect ('oversquashing') can hinder GNN performance.

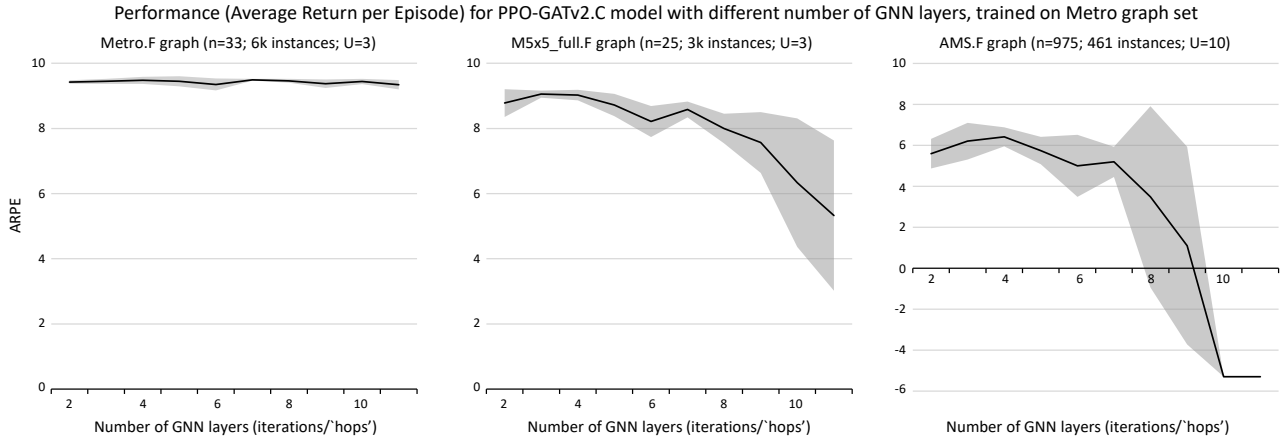


Figure E.2: Impact of increasing the number of GNN layers on escape model performance

We study this effect in our Search&Pursuit-Evasion setting. We train a PPO-GATv2 model on the Metro trainset (details in Appendix D). We vary the number of GNN layers in the escape agent model between 2-11 layers. The test performance on different graph sets is shown in Fig.E.2. Test performance on the trainset (left panel) remains constant for different GNN depths. Apparently, for this simple task, two GNN layers suffice and no oversquashing is observed. However, if we test on other (unseen) graph sets, test performance collapses for GNNs with more than 6-7 layers. Here we see the effect of the stagnation of information propagation due to bottleneck nodes in the graphs. Based on this observation, we limit the number of GNN layers to a maximum of five for the remainder of our experiments.

2. Effect of LSTM positioning

There are several options when including an LSTM module in our GNN-based policy approximation function. We test four options, depicted in Fig.E.3. The position of the LSTM in the computational graph can be varied from early stage (position number 1, acting on the node features), middle state (position 2, acting on the node embeddings) to late stage (position numbers 3/4, using separate LSTMs for Actor and Critic). In the late stage position, the LSTM can either act on the node embeddings $\boldsymbol{h}_v$, $v \in \mathcal{V}$, or on the concatenated vector of graph embedding and node embeddings $\left[\frac{1}{n}\sum_{w\in\mathcal{V}} \boldsymbol{h}_w \middle\| \boldsymbol{h}_v\right]$, similar to the use in Eq.5.8.

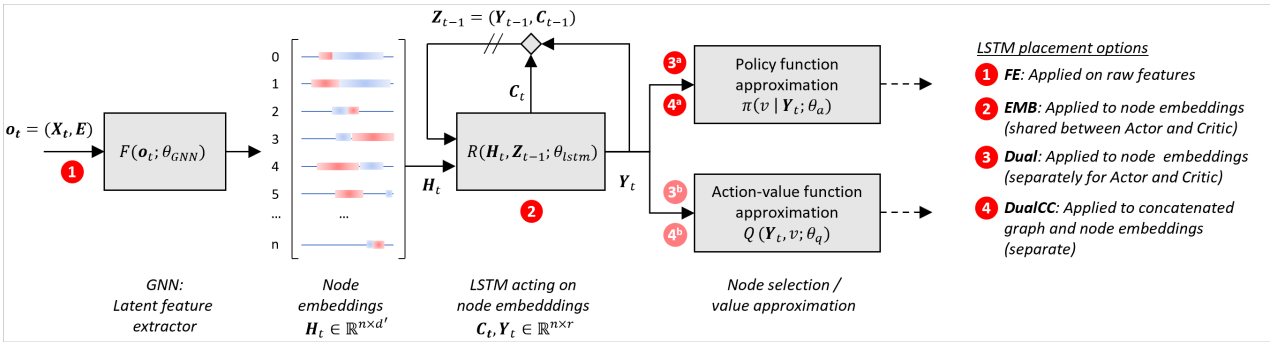


Figure E.3: LSTM placement options in GNN-based Actor-Critic models

In order to test our implementation, and experiment with these different LSTM placement options, we define a simple Search Pursuit-Evasion scenario, shown in Fig.E.4. Here, the escape agent (in red) starts on node 3 and aims to reach any of the two target nodes 2,5. At time $t = 0$, the escaper observes a pursuit agent, that is initialized either on on node 0 (as shown in the Figure) or on node 6 (the alternative initial graph state). At time $t = 1$, the escaper has no visibility: the pursuers position is masked and not included in the node feature matrix. If no memory is used, the escaper cannot distinguish the two initial graph states and has no way to choose a best action at $t = 1$. However, with memory, the pursuer presence at $t = 0$ can be remembered, and used to take a best action from node 4, away from the risk, so down to node 5 in the example of Fig.E.4.

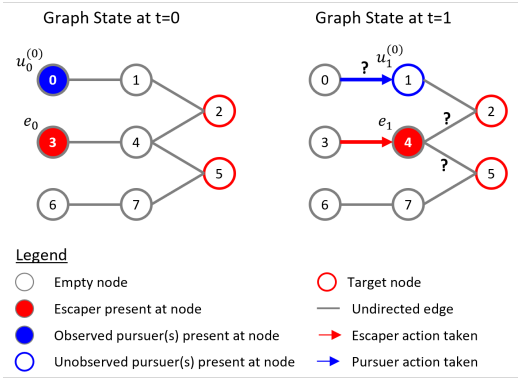


Figure E.4: MemTask definition

We train four variants of our baseline PPO-GNN-LSTM model with different LSTM positions (details can be found in Fig.D.5, model sizes range from ca. 20k to 60k parameters) with 600 iterations. This is clearly highly over-parametrized for this basic task, but we are interested in the general behavior of a baseline model that we could deploy on larger graphs and more complex scenarios. In addition to these four LSTM variants, we also train two models that have no LSTM, with one of them using frame-stacking of the input (with k=3 past frames stored in the node features). The resulting learning curves are shown in Fig.E.5.

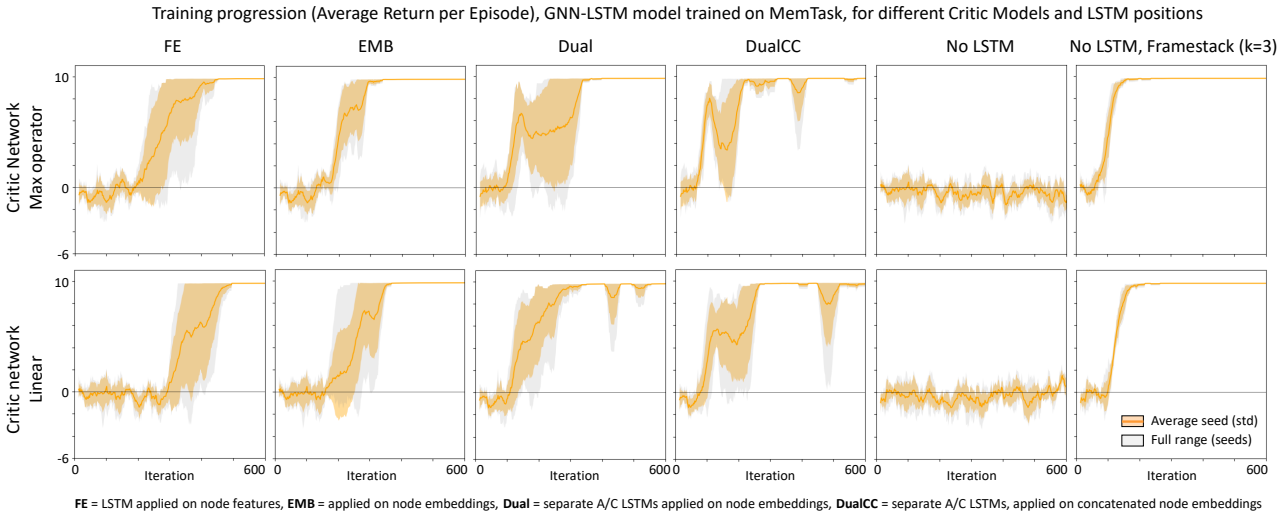


Figure E.5: LSTM performance analysis

As expected, if no memory is used (the 'No LSTM' chart), the optimal return cannot be obtained for this task. However, all other variants that include memory converge to the optimal solution. Different patterns can be observed in the learning curves of the different model variants. Two options stand out with fast convergence and low variance over seeds: the model with the LSTM acting on the node embeddings (denoted as 'EMB') and the model without an LSTM but with frame-stacking. Based on this result, we use these two variants in the remaining experiments.

3. Effect of increasing the number of pursuers during testing

To investigate how the trained baseline escape agent performs when the number of pursuers on the graphs is increased, we evaluate the PPO-GATv2-LSTM model, trained on the AMS trainset with U=10 pursuers, on three test graphs: AMS.F (part of the trainset), UTR.F and ROT.F (both not part of the trainset). Fig.E.6 shows the result, with the number of pursuers indicated on each row.

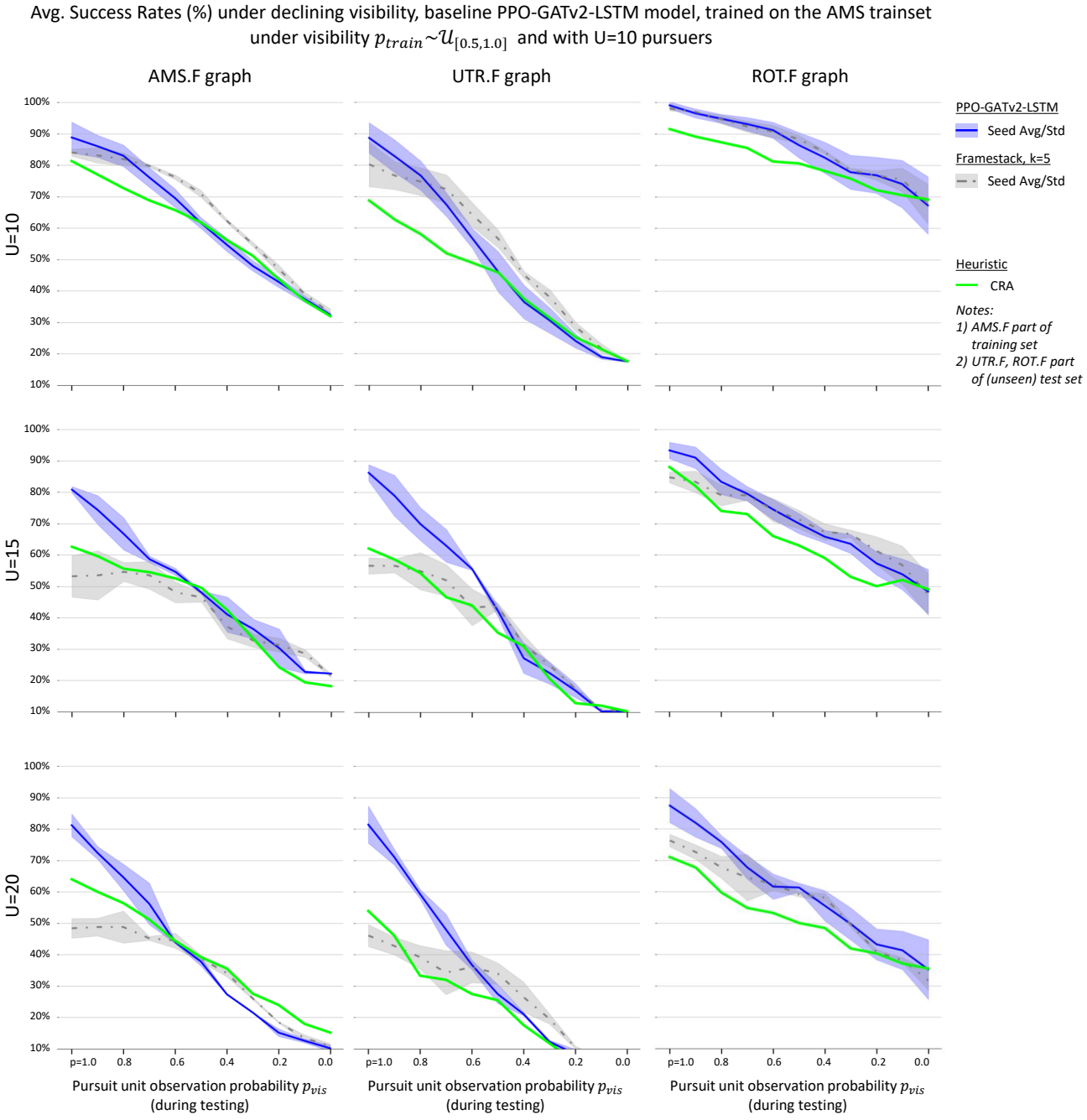


Figure E.6: Impact of increasing the number of pursuers on test graphs.

The main conclusion that can be drawn is that the baseline model with the LSTM retains a similar performance profile when the number of pursuers is increased. In the higher visibility region, it consistently outperforms the CRA benchmark. Interestingly, the model that uses frame-stacking instead of an LSTM cannot generalize to higher pursuer numbers; its performance drops towards (sometimes even below) the CRA benchmark.

4. Changing pursuer strategy

In section 6.4, we studied the effect of changing pursuer strategy on the performance of a trained escape agent. For completeness, we present the inverse scenario. Our baseline escape agent model is trained on the AMS trainset under pursuer response function $\mathscr{I}_{alt}$ and tested under $\mathscr{I}_{base}$. The result is shown in Fig.E.7 (orange line), and compared to the case where the escape agent was trained on $\mathscr{I}_{base}$ (blue line). We observe a similar effect to the effect shown in section 6.4. If tested under a different pursuer response function, the escape agent's performance will deteriorate. The reason for this is that it will not have been able to learn to predict (and hence, exploit) the behavioral patterns of the pursuit agents.

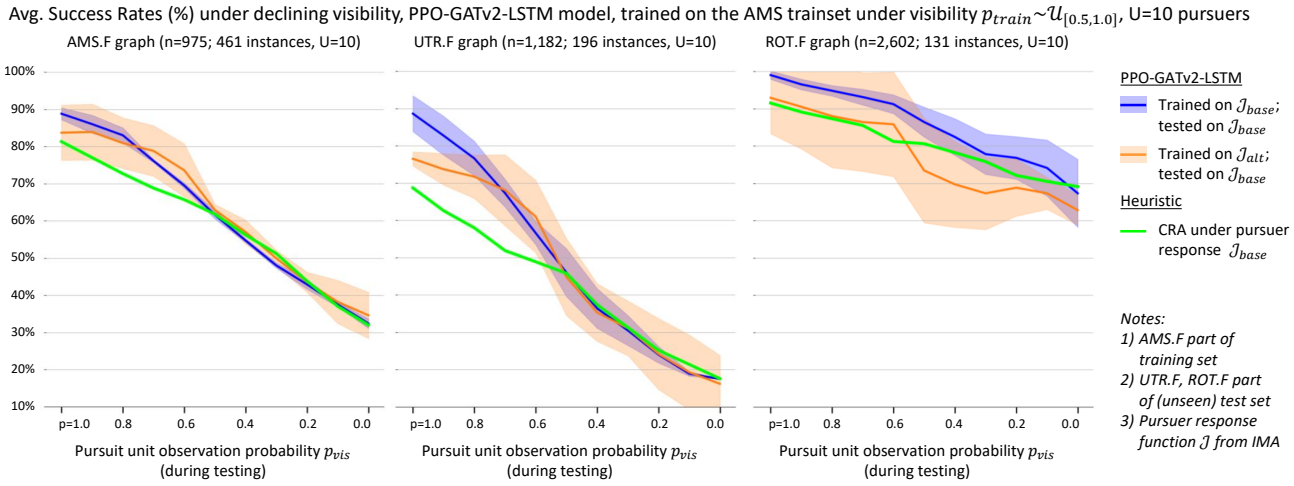


Figure E.7: Impact of varying pursuer response function $\mathscr{I}$ on test performance

5. Baseline performance using the ARPE metric

In order to verify whether the Success Rate (SR) metric correlates to the Average Return per Episode metric (ARPE), the baseline performance chart 6.5 used in section 6.3 is expressed in terms of ARPE in Fig.E.8. Indeed, the relative performance patterns are equivalent.

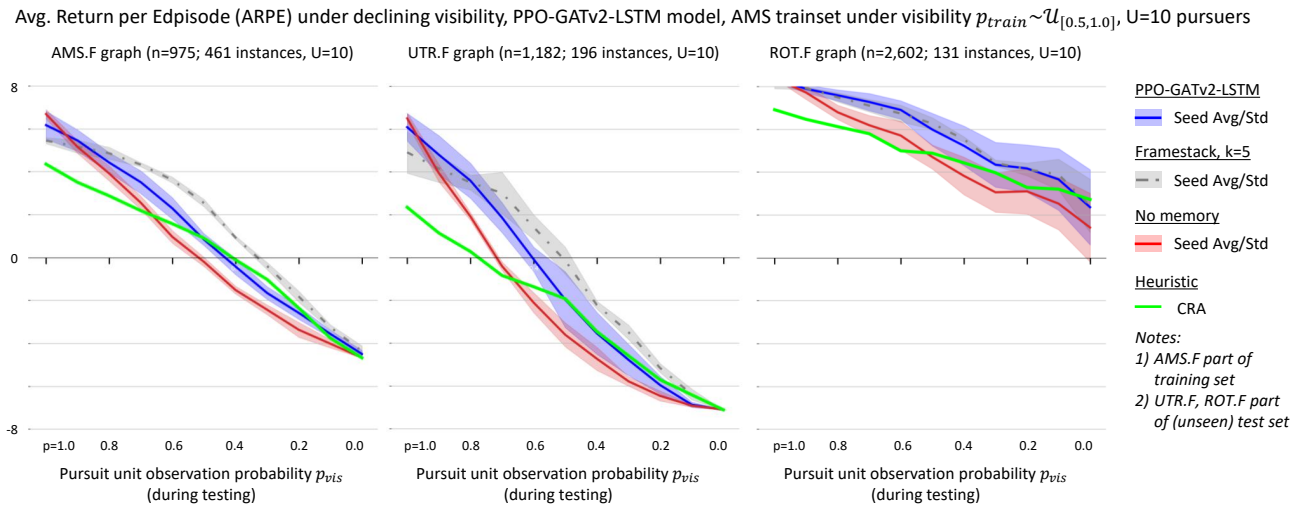


Figure E.8: Baseline test performance expressed in Average Return per Episode (ARPE)