

Memory Assignment for Finite-Memory Strategies in Adversarial Patrolling Games

Vojtěch Kůr, Vít Musil and Vojtěch Řehák*

Masaryk University, Faculty of Informatics, Brno, Czech Republic

Abstract. *Adversarial Patrolling games* form a subclass of Security games where a *Defender* moves between locations, guarding vulnerable targets. The main algorithmic problem is constructing a strategy for the Defender that *minimizes* the worst damage an *Attacker* can cause. We focus on the class of *finite-memory* (also known as regular) Defender’s strategies that experimentally outperformed other competing classes. A finite-memory strategy can be seen as a positional strategy on a finite set of states. Each state consists of a pair of a location and a certain integer value—called memory. Existing algorithms improve the transitional probabilities between the states but require that the available memory size itself is assigned at each location *manually*. Choosing the right memory assignment is a well-known open and hard problem that hinders the usability of finite-memory strategies. We solve this issue by developing a general method that iteratively changes the memory assignment. Our algorithm can be used in connection with *any* black-box strategy optimization tool. We evaluate our method on various experiments and show its robustness by solving instances of various patrolling models.

1 Introduction

This work follows the *patrolling games* line of work studying non-cooperative games where a mobile Defender guards resources against an Attacker. In *adversarial* patrolling [33, 2, 3, 9, 8, 14, 30], the Attacker *knows* the Defender’s strategy and observes his locations. The solution concept is based on *Stackelberg equilibrium*. The Defender commits to a strategy σ , and the Attacker chooses a counter-strategy π that maximizes the expected Attacker’s utility against σ . Intuitively, the value $\text{Val}(\sigma)$ denotes the damage caused by the best Attacker, and the precise definition of Val depends on the underlying *patrolling model*.

The environment where the agents play can be described by a Markov decision process. If the environment is not fully known, the problem becomes one of reinforcement learning, where the agent learns the strategy through interaction with the environment, using either model-free (e.g., Q-learning) or model-based (e.g., using a learned model to simulate outcomes) approaches. We focus on the cases when the *environment is fully known* both to the Defender and the Attacker. An example of such an environment is an oriented graph that typically describes the topology of the terrain. This concept is also used in the domain of planning.

Another aspect is the time frame in which the game is evaluated, which plays a critical role in shaping the agent’s behavior. We focus on an *infinite-horizon model*, where the Attacker can wait as long

as needed to exploit the best opportunity that maximizes the targets’ damage. This is especially suitable for uninterrupted services where the Defender represents many consecutive patrol shifts or a robot that is expected to run for a long time.

For general topologies, the existence of the optimal Defender’s strategy is PSPACE-complete [19]. Additionally, determining an ε -optimal strategy for $\varepsilon \leq 1/2n$, where n represents the number of vertices, is NP-hard [23]. Therefore, no polynomial algorithm can guarantee (sub)optimality in general (unless $P = NP$). Finding suitable heuristics applicable to real-world scenarios is of great importance.

Prior works focus on constructing *positional strategies* where the Defender makes a randomized choice of the next location based only on the current location [9, 5, 10, 13]. However, positional strategies fail to provide optimal protection even in a simple setting; see Figure 3 or [26].

To increase the strategy expressivity, Kučera and Lamser [26] introduced a class of *regular strategies*, where the Defender is equipped with a suitable finite-state automaton. In each location, the distribution over the next locations is determined by the state of the automaton after feeding it with the current history of locations. The authors proved that regular strategies offer better protection than positional strategies, even if the positional strategy depends on the last k locations. They also present a strategy synthesis algorithm for regular strategies. However, this approach’s main limitation is that the finite-state automaton must be supplied manually to achieve strong strategies.

The idea of regular strategies was followed by *finite-memory* strategies where the finite-state automaton is abstracted into finite integer memory denoted M [22]. The Defender chooses the new location based on the current location and memory value and also selects (possibly randomly) the new memory value.

Finite-memory strategies can also be seen as positional strategies on the set of states consisting of the pairs of the location and a memory integer between 1 and M . In this view, the state space increases dramatically, limiting the usability and performance of optimization algorithms. In fact, the size of the memory does not necessarily need to be the same in each location, i.e., the states can be set as $C = \{(v, i) \mid v \in V, 1 \leq i \leq \text{mem}(v)\}$, where V is a set of locations and $\text{mem}: V \rightarrow \mathbb{N}$ is certain *memory-assignment* function.

Recent works provide well-performing optimization algorithms for finite-memory strategies and various patrolling models [22, 24, 12, 25]. However, they all assume that a suitable memory assignment is given as an input. Currently, this is the main bottleneck of these algorithms, and the results depend heavily on the choice of mem .

* Corresponding Author. Email: xrehak@fi.muni.cz

All the published experiments either use uniform memory assignment with increasing M (that hits the limits of the algorithms already for $M \lesssim 10$) or use carefully handcrafted assignment mem based on the expert knowledge of the problem at hand. To the best of our knowledge, *no approach automates memory assignment* for general topologies. Klaška et al. [24] state that choosing the optimal set of states is an open and hard problem. We demonstrate the problem of optimal memory assignment in Example 1 below.

Example 1 Consider a graph where three target vertices v_1, v_2, v_3 are connected to a central vertex M , forming a star topology.

Assume that moving along any edge takes 1 time unit, and each target requires 6 time units to complete an attack. The Defender uses a patrolling strategy that cycles through the targets in the sequence:

$$v_1 \rightarrow M \rightarrow v_2 \rightarrow M \rightarrow v_3 \rightarrow M \rightarrow v_1.$$

Since the length of this cycle is 6, the patrol takes 6 time units to complete. This way, the Defender reaches each target within the attack duration and can thus intercept any ongoing attack in time.

However, this strategy is not positional since the Defender’s decision at M depends on the history. We can model this behavior as a finite-memory strategy, where the Defender must maintain three distinct internal states at M , denoted M_1, M_2, M_3 , resulting in the sequence:

$$v_1 \rightarrow M_1 \rightarrow v_2 \rightarrow M_2 \rightarrow v_3 \rightarrow M_3 \rightarrow v_1.$$

Now suppose that the attack duration of v_1 is 4 and the attack durations of v_2 and v_3 are 8. The original strategy is no longer good, since an observing and patient Attacker can wait for the moment when the Attacker moves from v_1 to M and initiate an attack on v_1 . Since the Defender will return to v_1 after 6 time units, the attack will be completed.

In this case, the optimal strategy returns to v_1 after each visit of v_2 and v_3 , resulting in a sequence:

$$v_1 \rightarrow M \rightarrow v_2 \rightarrow M \rightarrow v_1 \rightarrow M \rightarrow v_3 \rightarrow M \rightarrow v_1.$$

Here, the Defender needs two different states for v_1 and 4 for M . Both settings and their respective optimal strategies are visualized in Figure 13.

This example demonstrates the difficulty of choosing the optimal memory assignment. It shows that the minimal optimal set of states does not solely depend on the topology but differs based on particular properties of the targets. Moreover, memory may serve purposes beyond merely remembering the last visited vertex; it can capture more complex patterns in the Defender’s behavior.

Our Contribution

We propose a smart general heuristic algorithm that dynamically adjusts the memory assignment for finite-memory strategies without any expert knowledge. Our method works with any differentiable value function and any black-box strategy optimization tool. In combination, our memory assignment algorithm, together with any strategy synthesis algorithm, yields a general and fully automated method for a strong finite-memory Defender’s strategy.

We evaluate our method on several benchmarks and for various patrolling models. Our automated approach outperforms the manual memory assignment when mem is chosen to be uniform with increasing M in nearly all cases. Furthermore, our method is on par with expertly handcrafted memory assignments.

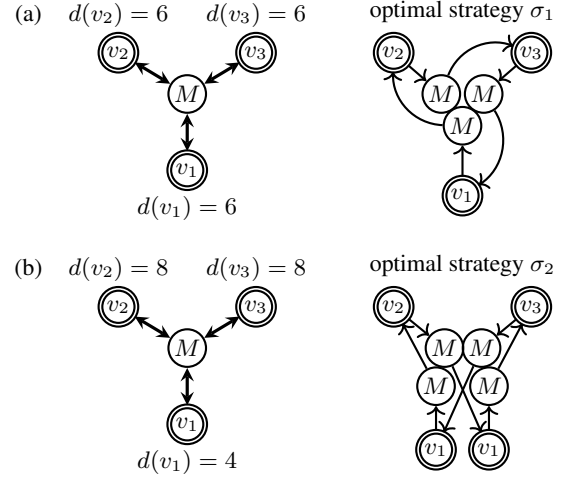


Figure 1. The optimal deterministic strategies require different memory allocations. The goal is to patrol a graph (a) with three targets and an internal location M . It takes $d = 6$ time units to complete an attack on each target. The optimal strategy is to cycle around the graph since the cycle’s length is 6. This is represented as σ_1 , which requires 3 different states at M . In (b), the attack durations change. The strategy σ_2 achieves perfect protection since v_1 is always visited every 4 steps, and both v_2 and v_3 are visited once on the cycle of length 8. Strategy σ_2 needs 4 states in M and 2 states in v_1 .

Related Work

This work contributes to the field of security games, which focuses on the optimal allocation of limited security resources to achieve effective target coverage, see monograph [31]. Patrolling games are a specialized type of security games where the Defender is mobile; see survey works [20, 4, 28, 7]. Most existing patrolling models can be categorized as either *regular* or *adversarial*.

Regular patrolling [1, 18, 15] is akin to surveillance, where the Defender’s goal is to quickly discover incidents by minimizing the time between consecutive visits to each target. In this model, the Defender typically employs a strategy involving a single path or cycle that visits all targets.

In contrast, *adversarial patrolling* [33, 2, 3, 9, 8, 14, 30, 7] focuses on protecting targets from an Attacker who seeks to exploit the best opportunities to maximize damage. This model is generally framed within the context of Stackelberg equilibrium [36, 29]. The Defender’s strategies are often *randomized* to prevent the Attacker from predicting future moves, and the Defender aims to maximize the probability of detecting an attack. The adversarial approach is particularly relevant in scenarios where a certain level of protection must be maintained, even if incidents occur at the most inconvenient times.

For patrolling scenarios of bounded duration, such as an eight-hour shift of a human ranger, *finite-horizon* models are sufficient. In contrast, *infinite-horizon* models are used when the duration is potentially unbounded or the bound is large, like 24/7 surveillance.

The model can also be distinguished by prior knowledge about the environment, which can be either fully known to all players or needs to be discovered during patrolling.

The difference between models has a substantial impact on strategy synthesis techniques. For finite horizon models with known environments, mathematical programming is the primary technique [9, 8]. Reinforcement learning has been largely successful for patrolling scenarios with a finite horizon and unknown environment, such as green security games [34, 11, 35, 21]. Gradient descent meth-

ods for finite-memory strategies are used extensively for infinite horizon models with known environments [22, 24, 12, 25].

Practical applications of security games include the deployment of police checkpoints at Los Angeles International Airport [27], the scheduling of federal air marshals on domestic airline flights across the U.S. [32], and the strategic arrangement of city guards in the Los Angeles Metro [16], positioning of U.S. Coast Guard patrols to secure critical locations [6], as well as the initiatives for wildlife protection in Uganda [17].

2 Background

First, we recall the standard notions of a patrolling graph, the Defender's and Attacker's strategy, and their value, and we fix the notation. Then, we formulate the requirements of our algorithm for the underlying patrolling model and show how existing patrolling models fit our framework. We assume familiarity with basic notions of calculus, linear algebra, probability theory, and finite-state discrete-time Markov chains.

2.1 Patrolling Graph

A *patrolling graph* is a tuple $G = (V, T, E, \text{time})$, where V is a non-empty set of *locations* (admissible Defender's positions), $T \subseteq V$ is a non-empty set of *targets*, $E \subseteq V \times V$ is a set of *edges* (admissible Defender's moves) and $\text{time}: E \rightarrow \mathbb{N}$ specifies the traversal time of an edge.

2.2 Defender's Strategy

In general, the Defender can choose the next location randomly based on the whole history of previously visited locations. However, general strategies may not be finitely representable. We focus on finite-memory Defender's strategies, which were shown to achieve the same limit protection as general strategies in two different patrolling models; see [24] and [25].

In finite-memory strategies, the Defender is equipped with an integer variable M called *memory*. A *state* of the Defender is a pair $(v, i) \in V \times \mathbb{N}$. A finite-memory strategy of the Defender can be seen as positional on a fixed finite set of states, i.e., it forms a discrete-time Markov chain on $V \times \mathbb{N}$.

To prevent the state space from blowing up, different amounts of memory can be allocated to each location. Hence, we define the *memory assignment* as a function $\text{mem}: V \rightarrow \mathbb{N}$ and a set of Defender's states

$$C = \{(v, i) \mid v \in V, 1 \leq i \leq \text{mem}(v)\}. \quad (1)$$

A finite-memory strategy of the Defender on V is a positional strategy on C , i.e., a function $\sigma: C \times C \rightarrow [0, 1]$ satisfying $\sum_{y \in C} \sigma(x, y) = 1$, for each $x \in C$, and $(u, v) \in E$, whenever $\sigma((u, i), (v, j)) > 0$. Intuitively, if the Defender is currently in a location u with a memory value i , the next state (v, j) is chosen with probability $\sigma((u, i), (v, j))$, which means that the Defender changes the memory value to j and starts traveling from u to v . The second condition ensures that the strategy respects the edges of the graph.

For every finite sequence of states $h = (c_1, \dots, c_n)$, we use $\mathbb{P}^{\sigma, c}[h]$ to denote the probability of executing h when the Defender starts patrolling in the state $c \in C$ and follows the strategy σ . That is, $\mathbb{P}^{\sigma, c}[h] = 0$ if $c_1 \neq c$, and $\mathbb{P}^{\sigma, c}[h] = \prod_{i=1}^{n-1} \sigma(c_i, c_{i+1})$ otherwise.

Strategy Types If $\text{mem} \equiv 1$, then we say that σ is positional on V or *memoryless*. If for every state x there is a state y such that $\sigma(x, y) = 1$, then we say that σ is *deterministic*. Note that σ can be both memoryless and deterministic.

2.3 Attacker's Strategy

In the patrolling graph, the time is spent traversing the edges. In adversarial models, the Attacker is assumed to perfectly observe the Defender's moves and can determine the next edge taken by the Defender immediately after its departure. For the Attacker, this is the best moment to attack because delaying the attack could only lower the attack's gain. Furthermore, the Attacker is allowed to attack only once in a single run.

An *observation* is a sequence of states $o = (c_1, \dots, c_n, c_{n+1})$. Intuitively, c_1 is the initial state of the Defender, c_n is the current state of the Defender, and c_{n+1} is the state chosen as the next one according to the Defender's strategy. The set of all observations is denoted by Ω . Formally, an *Attacker's strategy* is a function $\pi: \Omega \rightarrow T \cup \{\text{wait}\}$. Since the Attacker is allowed to attack only once, it is required that if $\pi(o) \in T$, then $\pi(o') = \text{wait}$ for every prefix o' of o .

2.4 Protection Value and Target Types

Let σ be a finite-memory Defender's strategy and π an Attacker's strategy. Let us fix an initial state c where the Defender starts patrolling. The expected damage the Attacker causes by attacking τ after observing o is denoted as $\mathcal{D}(o, \tau \mid \sigma)$. We will now present some examples of the definition of the value $\mathcal{D}(o, \tau \mid \sigma)$.

Hard-constrained targets One way of defining the damage on targets is to model a scenario where the Attacker needs to perform some action (e.g. picking a lock) to successfully complete an attack. If the target is visited in time, the Defender 'catches' the Attacker, and the damage is zero. Otherwise, the Attacker 'steals' the cost of the target. Here, $d(\tau)$ denotes the time it takes to complete an attack on τ and $\alpha(\tau)$ its cost. The value $\mathcal{D}(o, \tau \mid \sigma)$ is then defined as the probability that the Defender does not visit τ in the next $d(\tau)$ time units multiplied by $\alpha(\tau)$. We call this type of target *hard-constrained*. Within the context of finite-memory strategies, this model was first introduced by [22] and later used in [24, 12].

Blind targets The hard-constrained model can be extended by introducing a level of uncertainty where the ongoing attack is discovered only with probability $\beta(\tau)$ upon each visit of the target. We call these targets *blinded* and appear in [24, 12].

Linear targets Another target type models a situation where the actual damage depends on the time elapsed since initiating the attack till the Defender's visit, e.g., a fire or punching a hole in a fuel tank. Here, the target τ is assigned a value $v(\tau)$, which denotes the damage caused for every time unit while the target is under attack. Hence, the expected damage $\mathcal{D}(o, \tau \mid \sigma)$ equals the expected time it takes to visit τ multiplied by $v(\tau)$. If the probability of visiting τ after o is zero, then $\mathcal{D}(o, \tau \mid \sigma) = \infty$. We call this type of targets *linear*. It was introduced in [25].

2.5 Game Value

The *expected damage* caused by the Attacker is defined as

$$\text{Val}(\sigma, c, \pi) = \sum_{o \in \Omega, \pi(o) \in T} \mathbb{P}^{\sigma, c}[o] \cdot \mathcal{D}(o, \pi(o) \mid \sigma). \quad (2)$$

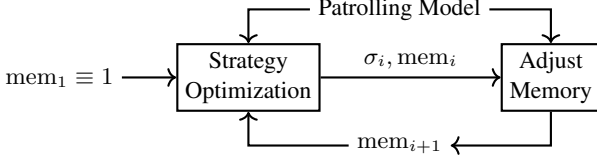


Figure 2. Diagram of our method. We iteratively run a strategy optimization black-box followed by our ADJUSTMEMORY procedure that updates the memory assignment.

The Defender/Attacker aims to minimize/maximize the damage, and hence, we define

$$\text{Val}(\sigma) = \min_{c \in C} \text{Val}(\sigma, c) = \min_{c \in C} \sup_{\pi} \text{Val}(\sigma, c, \pi). \quad (3)$$

By $\text{Val}(G)$, we denote the limit value of the best Defender's strategy or formally $\text{Val}(G) = \inf_{\sigma} \text{Val}(\sigma)$.

2.6 Requirements on the Patrolling Model

For our algorithm, we require two properties of the underlying patrolling model.

Let σ be a Defender's strategy and let $BSCC(\sigma)$ denote all subsets of states corresponding to bottom strongly connected components of the Markov chain determined by σ . For every $B \in BSCC(\sigma)$, let us denote by $\text{Val}(\sigma | B)$ the value

$$\max_{\tau \in T, e \in B \times B, \sigma(e) > 0} \mathcal{D}(e, \tau | \sigma).$$

Our first requirement is that $\text{Val}(\sigma)$ can be expressed as

$$\text{Val}(\sigma) = \min_{B \in BSCC(\sigma)} \text{Val}(\sigma | B). \quad (4)$$

Note that if σ is irreducible, then $\text{Val}(\sigma) = \text{Val}(\sigma | C)$.

Secondly, given a state space C , we require that for every $e \in C \times C$ and $\tau \in T$ the value $\mathcal{D}(e, \tau | \sigma)$ can be effectively calculated, and that $\mathcal{D}(e, \tau | \sigma)$ is a differentiable function with respect to σ whose gradient can also be evaluated.

The proofs that hard-constrained, blinded, and linear targets are eligible for our general framework are provided in the works where they were presented. In general, it can be seen that a sufficient condition for (4) is that the damage to the targets depends only on the current state and the future. That is, for every observation $o = (c_1, \dots, c_n, c_{n+1})$ and target τ , it holds

$$\mathcal{D}(o, \tau | \sigma) = \mathcal{D}((c_n, c_{n+1}), \tau | \sigma). \quad (5)$$

See the Appendix for a proof.

3 The Method

Here, we describe our method for solving the problem of choosing a suitable memory assignment. Our method works with any black-box optimization tool that improves the transitional probabilities and with any patrolling model that fits our general framework. Recall that the algorithms introduced in [24, 12, 25, 22] are eligible instances of such black-boxes.

The algorithm is initialized with the memory assignment mem_1 that assigns 1 to each location. That is, we start with a memoryless strategy. Then, we run the optimization tool and collect the resulting strategy σ_1 . From this, we run the core procedure ADJUSTMEMORY that outputs a new assignment mem_2 . The memory assignment mem_2 is then used for initialization for the tool, and we collect a new strategy σ_3 . This process continues for as long as the strategy value improves, see Figure 2.

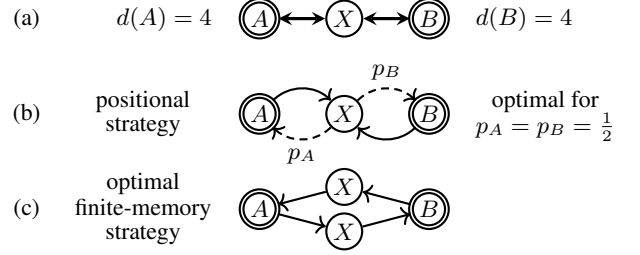


Figure 3. Patrolling on a graph (a) with three locations. End locations represent targets with costs of 1 and attack lengths equal to 4. Positional strategies (b) are parametrized by a single variable p . Due to symmetry, the optimality is reached for $p = 0.5$. The optimal strategy (b) is deterministic and needs two states in X .

3.1 Attacks Profiles

Now, we describe the core procedure ADJUSTMEMORY of our algorithm that decides how much memory should be added to which location. Intuitively, the procedure estimates the number of different 'behaviors' of each state that, on their own, could improve the current value.

Let us illustrate the idea with a simple example. Consider a patrolling graph depicted in Figure 3 with two targets A and B that are hard-constrained with $d = 4$ and $\alpha = 1$. We start with $\text{mem} = 1$. The state space is $C = \{(A, 1), (X, 1), (B, 1)\}$. For brevity, let us write v for $(v, 1)$ since the memory value is constant. In this example, from the end locations A and B , the Defender always returns to X , and hence the strategy depends only on complementary probabilities $\sigma(X, A)$ and $\sigma(X, B)$, which we denote by p_A and p_B , for short.

The best moment to attack A is when the Defender starts traversing from X to B . Then the probability of not visiting A in $d(A) = 4$ time units is exactly p_B and hence $\mathcal{D}((X, B), A | \sigma) = p_B$. Analogously, the best moment to attack B is when the Defender starts traversing from X to A and $\mathcal{D}((X, A), B | \sigma) = p_A$. The value $\text{Val}(\sigma)$ is then the maximum of both the attack values, i.e. $\text{Val}(\sigma) = \max\{p_A, p_B\}$ which is minimized when $\sigma(X, A) = \sigma(X, B) = 0.5$ with $\text{Val}(\sigma) = 0.5$.

We have reached the optimal value of positional strategies, and the question is whether the Defender can do better. The answer is positive since a finite deterministic strategy that moves from one end to the other and back represents a cycle of length 4. Hence, every attack is discovered in time, achieving the optimal value 0. This strategy can be represented by a finite-memory strategy on memory assignment mem_{det} with $\text{mem}_{\text{det}}(X) = 2$ and $\text{mem}_{\text{det}}(A) = \text{mem}_{\text{det}}(B) = 1$.

In practice, a reasonable optimization tool should output a strategy close to σ . Now, the key question is how to infer mem_{det} from σ . The idea is to consider gradients of all the maximal attacks w.r.t. the parameters of strategy σ . Any strategy σ can be modeled as Softmax of unconstrained parameters. In our example, we need just two real parameters, say (x_A, x_B) , so that

$$\text{Softmax}(x_A, x_B) = (p_A, p_B).$$

Using the chain rule, the partial derivatives of the maximal attacks w.r.t. the parameters x_A, x_B are

$$\frac{\partial \mathcal{D}((X, B), A | \sigma)}{\partial x_A} = -p_A p_B \quad \text{and} \quad \frac{\partial \mathcal{D}((X, B), A | \sigma)}{\partial x_B} = p_A p_B$$

and

$$\frac{\partial \mathcal{D}((X, A), B | \sigma)}{\partial x_A} = p_A p_B \quad \text{and} \quad \frac{\partial \mathcal{D}((X, A), B | \sigma)}{\partial x_B} = -p_A p_B.$$

We can see that the problem lies in the fact that for the attack on A , the gradient of the parameters is (up to a scale) $(-1, 1)$, while for the attack on B it is $(1, -1)$, which is in the exact opposite direction. Hence, the value can no longer improve since the competing gradients ‘cancel each other out’. This is exactly the place where we need to add a more memory value for X , one memory value to prioritize B (if the previous location was A) to cover the attack on B , and the second memory value to go to A (if the previous location was B) in order to cover the attack on A .

Formally, we fix a set of *eligible attacks* $\mathcal{E} \subseteq (C \times C) \times T$, corresponding to the attacks with close to maximal value. For an attack $(e, \tau) \in \mathcal{E}$ and a state $c \in C$, we consider the gradient of the attack value with respect to the outgoing probabilities of c

$$\nabla_{\sigma} \mathcal{D}(e, \tau \mid \sigma) = \left(\frac{\partial \mathcal{D}(e, \tau \mid \sigma)}{\partial \sigma(c, c_1)}, \dots, \frac{\partial \mathcal{D}(e, \tau \mid \sigma)}{\partial \sigma(c, c_n)} \right), \quad (6)$$

where $c_1, \dots, c_n$ are all the states with $\sigma(c, c_i) > 0$. Assuming that the strategy probabilities are parametrized by the Softmax of $x(c) = (x_1, \dots, x_n)$, the gradient of the attack value with respect to the strategy parameters is given by

$$\nabla_{x(c)} \mathcal{D}(e, \tau \mid \sigma) = \left(\frac{\partial \mathcal{D}}{\partial x_1}, \dots, \frac{\partial \mathcal{D}}{\partial x_n} \right) = J^T \cdot \nabla_{\sigma} \mathcal{D}(e, \tau \mid \sigma),$$

where J is the Jacobian of the Softmax. The *attack profile* of (e, τ) and c is then the *sign* of this vector $\nabla_{x(c)} \mathcal{D}(e, \tau \mid \sigma)$. We define profiles to be a function that assigns each state c the set of all (different) attack profiles belonging to c from $\mathcal{E}$. Formally,

$$\text{profiles}(c) = \{\text{sign}(\nabla_{x(c)} \mathcal{D}(e, \tau \mid \sigma)) \mid (e, \tau) \in \mathcal{E}\}.$$

The new memory assignment mem_2 is then defined as

$$\text{mem}_2(v) = \sum_{1 \leq i \leq \text{mem}_1(v)} |\text{profiles}(v, i)|. \quad (7)$$

The whole process is also described in the Algorithm 1. Here, the parameter $\varepsilon \in [0, 1)$ is a threshold for the set of eligible attacks. In our experiments, we consistently use $\varepsilon = 0.25$.

Time Complexity The time complexity of the Algorithm 1 is in

$$\mathcal{O}(f + |\mathcal{E}|g + |\mathcal{E}|^2|C|^2),$$

where f is such that the values of all attacks are calculated in $\mathcal{O}(f)$ and g is such that for each attack, the gradients with respect to the parameters are evaluated in $\mathcal{O}(g)$. Next, $|E|$ denotes the number of state edges such that its traversal probability is positive, and finally, $|\mathcal{E}|$ denotes the number of attacks taken into account.

Proof. In the algorithm, we keep a list of profiles for every state. First, we need to determine the values of all attacks in $\mathcal{O}(f)$. For every attack in the threshold, we calculate the gradients in $\mathcal{O}(g)$. Then, for each state, we calculate the sign of the vector in $\mathcal{O}(|C|)$ and decide if it’s already in the list. This gives at most $|\mathcal{E}|$ iterations with equality of profiles being in $\mathcal{O}(|C|)$. Determining the size of the profiles and summing over vertices is done in $\mathcal{O}(C)$. $\square$

In practice, the evaluation for hard-constrained and blinded was faster than for linear. This is primarily because the first algorithm’s core procedure is implemented in C++ with a dynamic program and uses a sparse representation of the strategy, unlike the second, which is in PyTorch and uses full matrices. In both cases, the actual bottleneck was in calculating the gradients.

Algorithm 1: Core procedure of our method.

```

Function AdjustMemory( $\sigma$ , mem,  $\varepsilon$ ):
    Define  $x(c)(c_i)$  to be  $\log(\sigma(c, c_i))$ 
    foreach  $c \in C$  do
        | profiles( $c$ )  $\leftarrow \emptyset$ 
    foreach  $e \in C \times C, \tau \in T$  do
        | if  $\mathcal{D}(e, \tau \mid \sigma) < (1 - \varepsilon) \text{Val}(\sigma)$  then
            | | continue
        | foreach  $c \in C$  do
            | | profile  $\leftarrow \text{sign}(\nabla_{x(c)} \mathcal{D}(e, \tau \mid \sigma))$ 
            | | Add profile to profiles( $c$ )
    foreach  $v \in V$  do
        | mem'( $v$ )  $\leftarrow 0$ 
        | for  $i \leftarrow 1$  to mem( $v$ ) do
            | | mem'( $v$ )  $\leftarrow \text{mem}'(v) + |\text{profiles}(v, i)|$ 
    return mem'

```

3.2 Modifications

We also introduce a modification of our method for the cases when the total number of states is bounded above by L . Bounding the maximal number of states of the strategy may be both desired by the user (memory of a robot) and the whole optimization process (algorithms run very slowly for strategies with a high number of states). Here, the idea is to assign value to each profile and then take at most L profiles with the highest value. For an exact description, see the Appendix.

Some optimization tools (like gradient-descent ones) enable us to warm-start the process instead of always starting from random parameters. In the Appendix, we present a way to transform a strategy σ_i on mem_i to σ_{i+1} on mem_{i+1} . However, we conclude that this does not provide a significant advantage, and we omit this extension from the experiments presented here.

4 Experiments

We evaluate our memory assignment method in a series of experiments with the best-performing algorithm for hard-constrained and blind targets of [12] and the algorithm for linear targets of [25]. We created a unified Python project containing all compatible state-of-the-art algorithms with existing relevant benchmarks. The repository will be publicly released (preliminarily, see the Appendix).

We also test the manual assignment *degree* that sets $\text{mem}(v)$ to the out-degree of v . This is precisely the memory needed when an Eulerian cycle forms the optimal strategy. This memory was expertly handcrafted by the authors [25] for their experiments.

We compare uniform memory assignments (denoted by a single number) with the degree assignment (*degree*), and our automatic method (*auto*). We ran experiments on three patrolling models/target types (hard-constrained, blind, linear) on over 50 different patrolling graphs (Offices, Building, Airports, Terrains, Stars).

For each setting, we executed 200 runs with a timeout of 180 seconds to get reliable statistical information. The number of states is bounded by 300. Recall that the whole procedure is stochastic as the optimization starts from a random init and may also contain non-deterministic updates.

Here, we present only a fragment of our experiments to highlight our key observations and limitations. The broader summary is in Section 4.1. The full report, detailed setup, and reproducibility steps can be found in the Appendix.

Offices This benchmark comes from [24]. The goal is to patrol office buildings with 1, 2, or 3 floors. The offices are designed so that

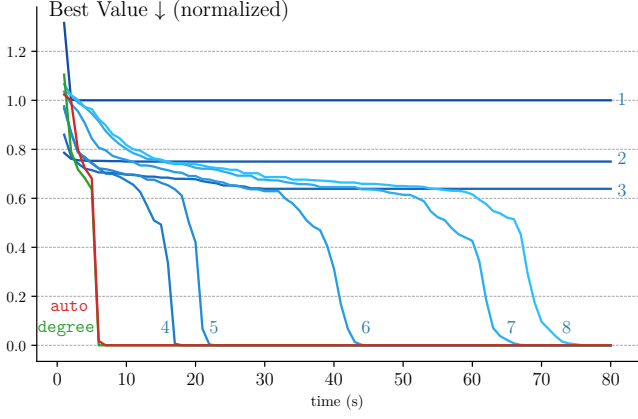


Figure 4. Values over time of the best performing strategies (out of 200 restarts) on one-floor **Offices**. The values are normalized by the value of the best memoryless strategy, optimum is zero. Uniform memory below 4 cannot achieve optimum; higher memory gets zero but significantly later as the memory increases. Both Expert (*degree*) and Automatic (*auto*) outperform the uniform assignment.

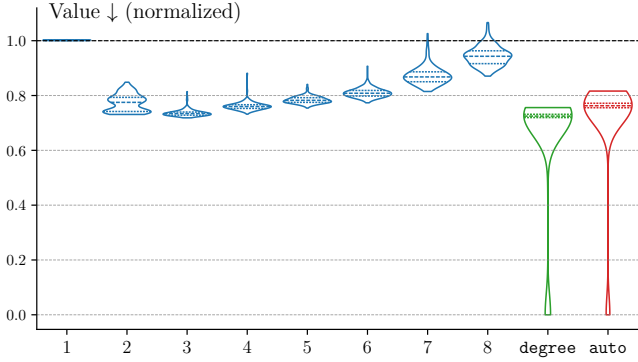


Figure 5. Comparison of memory assignments on the **Offices** benchmark (2 floors). Statistics are over 200 restarts. Uniform $\text{mem} = 4$ is sufficient for the optimal strategy, but never found for any of the uniform within the timeout. Both Expert (*degree*) and Automatic (*auto*) memory assignments lead to the optimum.

an Eulerian cycle is the optimal strategy. The maximum degree is 4; hence, the uniform memory must be at least 4 to achieve optimum. Figure 4 shows the optimization dynamics on 1 floor and Figure 5 reports the values of strategies for offices on 2 floors.

Building [24] also used the same topology as **Offices** to evaluate the synthesis of blinded targets. We recreate this benchmark, and the results can be found in the Appendix.

Airports This benchmark originated from [25]. The goal is to patrol airport gates. An airport with n halls has $3n + 1$ nodes, from which $2n$ are linear targets. In Figure 6, we report the distribution of strategy values for $n = 19$. We also ran the benchmark with random target values sampled uniformly and independently between 1 and 10.

Stars We introduce a new benchmark on star-shaped graphs, where the targets and the attack lengths are chosen so that the optimal strategy requires a nontrivial amount of memory. The graph is parametrized by a number of groups k ranging from 1 to 5. A graph

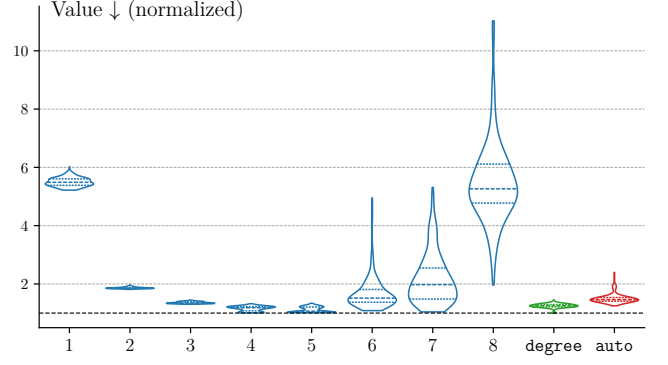


Figure 6. Comparison of memory assignments on the **Airport** with 19 halls. The values are normalized by the value of the baseline (Eulerian cycle requiring *degree* memory). Uniform memory (for $\text{mem} = 4$) is sufficient to meet the Expert (*degree*) assignment. This is one of the rare examples where the Automatic (*auto*) assignment lags behind.

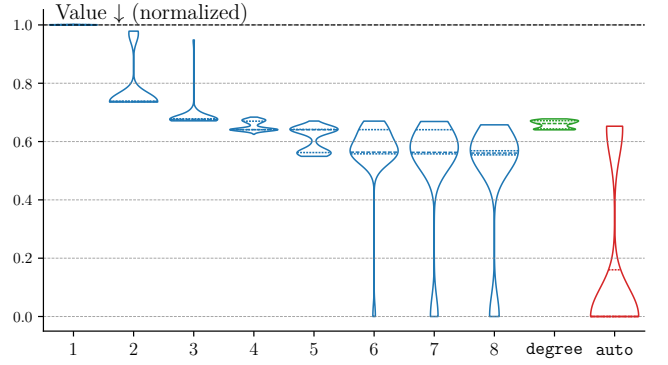


Figure 7. Comparison of memory assignments for the **Star** experiment with 3 groups. The values are normalized by the value of the best memoryless strategy found, optimum is zero. The optimal strategy requires more than a degree amount of memory. Automatic assignment achieves optimum consistently.

with k groups has $k + 2$ vertices. In Figure 7, we report the statistics for $k = 3$.

Terrains Another new benchmark consists of randomly generated connected planar graphs that model open terrains. The nodes are randomly generated on a plane, triangulated, connected by the minimum spanning tree, and each edge from triangulation is added with a probability of 0.5. We run the experiments with the odd number of nodes n ranging from 3 to 41. Figure 8 shows the statistics for $n = 33$.

4.1 Discussion

Failure of uniform We observe that uniform memory assignments are rarely optimal. Oftentimes, more memory is needed and low values of M achieve poor results. In theory, if there is an optimal finite-memory strategy, then a high enough uniform assignment suffices. The authors of [24] state that the probability of finding the optimal strategy increases with M in the **Offices** benchmark. We verify this, but higher M also significantly increases the state space and the time needed to find the optimal strategy (see Figure 4).

Therefore, within a fixed timeout, these two competing aspects (sizes of M and state space) create a convex-shaped dependency on

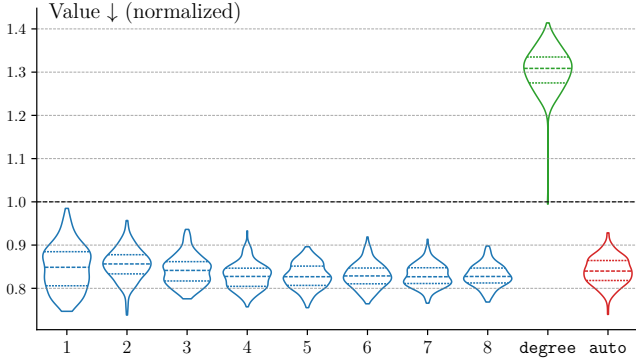


Figure 8. Comparison of memory assignments for the **Terrains** experiment with size $n = 33$. The values are normalized by a baseline (the best deterministic strategy). Larger memory does not help here. Degree assignment causes state space too large, and well-performing strategies are not found.

M , where the lowest values may or may not hit the optimum (cf. Figures 6 and 5). The behavior is hard to predict without intensive testing, making the uniform assignment impractical.

We can conclude that the distribution of memory states matters. Degree assignment proved to be a simple heuristic that often yields superior results. Automatic assignment achieves better or comparable results than uniform in almost all cases.

Limitations of degree For Offices and Airport benchmarks, degree memory assignment produced the best results. However, it scales poorly when it adds too much memory (Terrains, Building), see Figure 8. On the opposite, it may provide an insufficient amount of memory, see Figure 7. While it is a good heuristic for smaller graphs with a small maximal degree, it may achieve poor results on big or highly connected graphs or in cases when the optimal strategy requires more information than remembering the next vertex.

Robustness of auto Our proposed automatic assignment achieves high robustness across all the experiments. This heuristic enables the user to fully utilize the power of the synthesis tool. On the one hand, we can push the tool to find complex strategies requiring memory (Offices, Airports, Stars). On the other hand, we keep memory low if it is not needed, enabling the tool to converge to a high-quality strategy (Terrains). Our method also scales well (Building).

A notable exception is the slight under-performance on the Airport benchmark, where for larger graphs, our automatic method achieves worse results than degree, see Figure 6. The reason is that this benchmark uses linear targets whose objective function is not bounded, and the convergence of positional strategies is slow.

We hypothesize that modifying the linear objective so that it is bounded would lead to better performance of auto, even for larger graphs. The advantage of our approach is that our heuristic would be usable for this modification as well.

5 Conclusion

We proposed an automatic memory assignment method for patrolling problems and compared it against uniform and expert, degree-based assignments. Our experiments across diverse benchmarks showed that our automatic method outperformed uniform memory in nearly all cases, offering robust and scalable performance.

Our approach provides a versatile and novel heuristic that is independent of the underlying optimization algorithms and objectives,

making it applicable to a wider class of patrolling-type problems and new optimization techniques.

References

- [1] P. Afshani, M. de Berg, K. Buchin, J. Gao, M. Löffler, A. Nayyeri, B. Raichel, R. Sarkar, H. Wang, and H. Yang. Approximation algorithms for multi-robot patrol-scheduling with min-max latency. In *Proceedings of WAFR 2021*, volume 17 of *Springer Proceedings in Advanced Robotics*, pages 107–123. Springer, 2021. doi: 10.1007/978-3-030-66723-8_7. URL https://doi.org/10.1007/978-3-030-66723-8_7.
- [2] N. Agmon, S. Kraus, and G. A. Kaminka. Multi-robot perimeter patrol in adversarial settings. In *Proceedings of IEEE 2008*, pages 2339–2345. IEEE, 2008. doi: 10.1109/ROBOT.2008.4543563. URL <https://doi.org/10.1109/ROBOT.2008.4543563>.
- [3] N. Agmon, S. Kraus, G. A. Kaminka, and V. Sadov. Adversarial uncertainty in multi-robot patrol. In *Proceedings of IJCAI 2009*, pages 1811–1817, 2009. URL <http://ijcai.org/Proceedings/09/Papers/301.pdf>.
- [4] A. Almeida, G. L. Ramalho, H. Santana, P. A. Tedesco, T. Menezes, V. Corruble, and Y. Chevaleyre. Recent advances on multi-agent patrolling, 2004. URL https://doi.org/10.1007/978-3-540-28645-5_48.
- [5] S. Alpern, P. Chleboun, S. Katsikas, and K. Y. Lin. Adversarial patrolling in a uniform. *Oper. Res.*, 70(1):129–140, 2022. doi: 10.1287/OPRE.2021.2152. URL <https://doi.org/10.1287/opre.2021.2152>.
- [6] B. An, E. Shieh, M. Tambe, R. Yang, C. Baldwin, J. DiRenzo, B. Maule, and G. Meyer. PROTECT - A deployed game theoretic system for strategic security allocation for the united states coast guard. *AI Mag.*, 33(4):96–110, 2012. doi: 10.1609/AIMAG.V33i4.2401. URL <https://doi.org/10.1609/aimag.v33i4.2401>.
- [7] N. Basilico. Recent trends in robotic patrolling. *Current Robotics Reports*, 3(2):65–76, 2022. ISSN 2662-4087. URL <https://doi.org/10.1007/s43154-022-00078-5>.
- [8] N. Basilico, N. Gatti, and F. Amigoni. Leader-follower strategies for robotic patrolling in environments with arbitrary topologies. In *Proceedings of AAMAS 2009*, pages 57–64. IFAAMAS, 2009. URL <https://dl.acm.org/citation.cfm?id=1558020>.
- [9] N. Basilico, N. Gatti, and F. Amigoni. Patrolling security games: Definition and algorithms for solving large instances with single patroller and single intruder. *Artif. Intell.*, 184-185:78–123, 2012. doi: 10.1016/J.ARTINT.2012.03.003. URL <https://doi.org/10.1016/j.artint.2012.03.003>.
- [10] N. Basilico, G. D. Nittis, and N. Gatti. Adversarial patrolling with spatially uncertain alarm signals. *Artif. Intell.*, 246:220–257, 2017. doi: 10.1016/J.ARTINT.2017.02.007. URL <https://doi.org/10.1016/j.artint.2017.02.007>.
- [11] A. Biswas, G. Aggarwal, P. Varakantham, and M. Tambe. Learn to intervene: An adaptive learning policy for restless bandits in application to preventive healthcare. In *Proceedings of IJCAI 2021*, pages 4039–4046. ijcai.org, 2021. doi: 10.24963/IJCAI.2021/556. URL <https://doi.org/10.24963/ijcai.2021/556>.
- [12] T. Brázdil, D. Kláška, A. Kučera, V. Musil, P. Novotný, and V. Řehák. On-the-fly adaptation of patrolling strategies in changing environments. In *Proceedings of UAI 2022*, volume 180 of *Proceedings of Machine Learning Research*, pages 244–254. PMLR, 2022. URL <https://proceedings.mlr.press/v180/brazdil22a.html>.
- [13] A. Collins, J. Czyzowicz, L. Gasieniec, A. Kosowski, E. Kranakis, D. Krizanc, R. Martin, and O. Morales-Ponce. Optimal patrolling of fragmented boundaries. In *Proceedings of SPAA 2013*, pages 241–250. ACM, 2013. doi: 10.1145/2486159.2486176. URL <https://doi.org/10.1145/2486159.2486176>.
- [14] E. M. de Cote, R. Stranders, N. Basilico, N. Gatti, and N. R. Jennings. Introducing alarms in adversarial patrolling games: extended abstract. In *Proceedings of AAMAS 2013*, pages 1275–1276. IFAAMAS, 2013. URL <http://dl.acm.org/citation.cfm?id=2485180>.
- [15] A. Farinelli, L. Iocchi, and D. Nardi. Distributed on-line dynamic task assignment for multi-robot patrolling. *Autonomous Robots*, 41(6): 1321–1345, 2017. URL <https://doi.org/10.1007/s10514-016-9579-8>.
- [16] F. M. D. Fave, A. X. Jiang, Z. Yin, C. Zhang, M. Tambe, S. Kraus, and J. P. Sullivan. Game-theoretic patrolling with dynamic execution uncertainty and a case study on a real transit system. *J. Artif. Intell. Res.*, 50:321–367, 2014. doi: 10.1613/JAIR.4317. URL <https://doi.org/10.1613/jair.4317>.
- [17] B. J. Ford, D. Kar, F. M. D. Fave, R. Yang, and M. Tambe. PAWS: adaptive game-theoretic patrolling for wildlife protection. In *Proceedings of AAMAS 2014*, pages 1641–1642. IFAAMAS/ACM, 2014. URL <https://dl.acm.org/citation.cfm?id=2616103>.

- [18] S. K. K. Hari, S. Rathinam, S. Darbha, K. Kalyanam, S. G. Manyam, and D. W. Casbeer. Optimal UAV route planning for persistent monitoring missions. *IEEE Trans. Robotics*, 37(2):550–566, 2021. doi: 10.1109/TRO.2020.3032171. URL <https://doi.org/10.1109/TRO.2020.3032171>.
- [19] H. Ho and J. Ouaknine. The cyclic-routing UAV problem is pspace-complete. In *Proceedings of ETAPS 2015*, volume 9034 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2015. doi: 10.1007/978-3-662-46678-0_21. URL https://doi.org/10.1007/978-3-662-46678-0_21.
- [20] L. Huang, M. Zhou, K. Hao, and E. S. H. Hou. A survey of multi-robot regular and adversarial patrolling. *IEEE CAA J. Autom. Sinica*, 6(4): 894–903, 2019. doi: 10.1109/JAS.2019.1911537. URL <https://doi.org/10.1109/JAS.2019.1911537>.
- [21] J. Karwowski, J. Mandziuk, A. Zychowski, F. Grajek, and B. An. A memetic approach for sequential security games on a plane with moving targets. In *Proceedings of AAAI 2019*, pages 970–977. AAAI Press, 2019. doi: 10.1609/AAAI.V33i01.3301970. URL <https://doi.org/10.1609/aaai.v33i01.3301970>.
- [22] D. Klaška, A. Kučera, T. Lamser, and V. Řehák. Automatic synthesis of efficient regular strategies in adversarial patrolling games. In *Proceedings of AAMAS 2018*, pages 659–666. IFAAMAS, 2018. URL <http://dl.acm.org/citation.cfm?id=3237481>.
- [23] D. Klaška, A. Kučera, and V. Řehák. Adversarial patrolling with drones. In *Proceedings of AAMAS 2020*, pages 629–637. IFAAMAS, 2020. doi: 10.5555/3398761.3398837. URL <https://dl.acm.org/doi/10.5555/3398761.3398837>.
- [24] D. Klaška, A. Kučera, V. Musil, and V. Řehák. Regstar: efficient strategy synthesis for adversarial patrolling games. In *Proceedings of UAI 2021*, volume 161 of *Proceedings of Machine Learning Research*, pages 471–481. AUAI Press, 2021. URL <https://proceedings.mlr.press/v161/klaska21a.html>.
- [25] D. Klaška, A. Kučera, V. Musil, and V. Řehák. Minimizing expected intrusion detection time in adversarial patrolling. In *Proceedings of AAMAS 2022*, pages 1660–1662. IFAAMAS, 2022. doi: 10.5555/3535850.3536068. URL <https://www.ifaamas.org/Proceedings/aamas2022/pdfs/pl1660.pdf>.
- [26] A. Kučera and T. Lamser. Regular strategies and strategy improvement: Efficient tools for solving large patrolling problems. In *Proceedings of AAMAS 2016*, pages 1171–1179. ACM, 2016. URL <http://dl.acm.org/citation.cfm?id=2937095>.
- [27] J. Pita, M. Jain, J. Marecki, F. Ordóñez, C. Portway, M. Tambe, C. Western, P. Paruchuri, and S. Kraus. Deployed ARMOR protection: the application of a game theoretic model for security at the los angeles international airport. In *Proceedings of AAMAS 2008*, pages 125–132. IFAAMAS, 2008. URL <https://dl.acm.org/citation.cfm?id=1402819>.
- [28] D. Portugal and R. P. Rocha. A survey on multi-robot patrolling algorithms. In *Technological Innovation for Sustainability - Second IFIP WG 5.5/SOCOLNET Doctoral Conference on Computing, Electrical and Industrial Systems, DoCEIS 2011, Costa de Caparica, Portugal, February 21-23, 2011. Proceedings*, volume 349 of *IFIP Advances in Information and Communication Technology*, pages 139–146. Springer, 2011. doi: 10.1007/978-3-642-19170-1_15. URL https://doi.org/10.1007/978-3-642-19170-1_15.
- [29] A. Sinha, F. Fang, B. An, C. Kiekintveld, and M. Tambe. Stackelberg security games: Looking beyond a decade of success. In *Proceedings of IJCAI 2018*, pages 5494–5501. ijcai.org, 2018. doi: 10.24963/IJCAI.2018/775. URL <https://doi.org/10.24963/ijcai.2018/775>.
- [30] E. Sless, N. Agmon, and S. Kraus. Multi-robot adversarial patrolling: Handling sequential attacks. *Artif. Intell.*, 274:1–25, 2019. doi: 10.1016/J.ARTINT.2019.02.004. URL <https://doi.org/10.1016/j.artint.2019.02.004>.
- [31] M. Tambe. *Security and Game Theory - Algorithms, Deployed Systems, Lessons Learned*. Cambridge University Press, 2012. ISBN 978-1-10-709642-4. URL <http://www.cambridge.org/de/academic/subjects/computer-science/communications-information-theory-and-security/security-and-game-theory-algorithms-deployed-systems-lessons-learned?format=AR>.
- [32] J. Tsai, S. Rathi, C. Kiekintveld, F. Ordóñez, M. Tambe, and T. Networks. Iris – a tool for strategic security allocation in transportation networks. *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS*, 2, 12 2011. doi: 10.1017/CBO9780511973031.005.
- [33] Y. Vorobeychik, B. An, and M. Tambe. Adversarial patrolling games. In *Proceedings of AAMAS 2012*, pages 1307–1308. IFAAMAS, 2012. URL <http://dl.acm.org/citation.cfm?id=2343977>.
- [34] Y. Wang, Z. R. Shi, L. Yu, Y. Wu, R. Singh, L. Joppa, and F. Fang. Deep reinforcement learning for green security games with real-time

information. In *Proceedings of AAAI 2019*, pages 1401–1408. AAAI Press, 2019. doi: 10.1609/AAAI.V33i01.33011401. URL <https://doi.org/10.1609/aaai.v33i01.33011401>.

- [35] L. Xu. Learning and planning under uncertainty for green security. In *Proceedings of IJCAI 2021*, pages 4927–4928. ijcai.org, 2021. doi: 10.24963/IJCAI.2021/695. URL <https://doi.org/10.24963/ijcai.2021/695>.
- [36] Z. Yin, D. Korzhik, C. Kiekintveld, V. Conitzer, and M. Tambe. Stackelberg vs. nash in security games: interchangeability, equivalence, and uniqueness. In *Proceedings of AAMAS 2010*, pages 1139–1146. IFAAMAS, 2010. URL <https://dl.acm.org/citation.cfm?id=1838360>.

A Proof of the Sufficient Condition

Generally, it can be seen that a sufficient condition on the first requirement of our framework is that the damage to targets depends only on the current state and the future. That is, for every observation $o = (c_1, \dots, c_n, c_{n+1})$ and target τ it holds

$$\mathcal{D}(o, \tau \mid \sigma) = \mathcal{D}((c_n, c_{n+1}), \tau \mid \sigma). \quad (8)$$

Proof. For the purposes of the proof, let us denote by M the expression $\min_{B \in BSSC(\sigma)} \text{Val}(\sigma \mid B)$. We prove that for every strategy σ on a set of states C it holds

$$\min_{c \in C} \sup_{\pi} \sum_{\substack{o \in \Omega \\ \pi(o) \in T}} \mathbb{P}^{\sigma, c}[o] \cdot \mathcal{D}(o, \pi(o) \mid \sigma) = M.$$

Also recall that the left-hand side is $\text{Val}(\sigma) = \min_c \text{Val}(\sigma, c) = \sup_{\pi} \text{Val}(\sigma, c, \pi)$.

First, we will show $\text{Val}(\sigma) \leq M$. It is sufficient to show that there is a state $c \in C$, such that $\text{Val}(\sigma, c) \leq M$. Let us choose any $B \in BSSC(\sigma)$ such that $\text{Val}(\sigma \mid B) = M$ and c as any state in B .

Now, we aim to show that $\text{Val}(\sigma, c, \pi) \leq \text{Val}(\sigma \mid B)$ for every Attacker’s strategy π . Let π be any strategy of the Attacker. Using (8) we can write that $\text{Val}(\sigma, c, \pi)$ equals

$$\sum_{\substack{o \in \Omega \\ o=(c_1, \dots, c_n, c_{n+1}) \\ \pi(o) \in T}} \mathbb{P}^{\sigma, c}[o] \cdot \mathcal{D}((c_n, c_{n+1}), \pi(o) \mid \sigma)$$

Since c lies in a bottom strongly connected component B , every observation $o = (c_1, \dots, c_n, c_{n+1})$ with positive $\mathbb{P}^{\sigma, c}[o]$ must have that each transition $e = (c_i, c_{i+1})$ is in $B \times B$ and $\sigma(e) > 0$. Namely $(c_n, c_{n+1}) \in B \times B$ and $\sigma(c_n, c_{n+1}) > 0$. From that it follows $\mathcal{D}((c_n, c_{n+1}), \pi(o) \mid \sigma) \leq \text{Val}(\sigma \mid B)$ for all observations with positive probability. Using this, we can write

$$\begin{aligned} \text{Val}(\sigma, c, \pi) &\leq \sum_{\substack{o \in \Omega \\ \pi(o) \in T}} \mathbb{P}^{\sigma, c}[o] \cdot \text{Val}(\sigma \mid B) \\ &= \text{Val}(\sigma \mid B) \sum_{\substack{o \in \Omega \\ \pi(o) \in T}} \mathbb{P}^{\sigma, c}[o]. \end{aligned}$$

The desired inequality follows from the fact that the sum of probabilities of the observations where the Attacker attacks is at most 1. Formally, it is true that

$$\sum_{\substack{o \in \Omega \\ \pi(o) \in T}} \mathbb{P}^{\sigma, c}[o] \leq 1,$$

which follows from the fact that the Attacker is allowed to attack only once in a run.

For the second inequality $\text{Val}(\sigma) \geq M$, it is sufficient to show that there is a strategy of the Attacker π such that $\text{Val}(\sigma, c, \pi) \geq M$ for every $c \in C$.

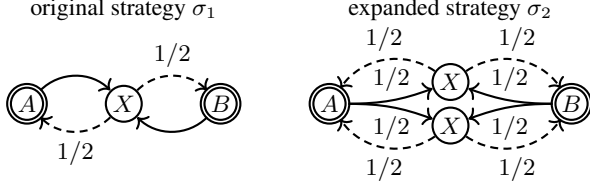


Figure 9. Demonstration of strategy expansion. The original strategy is σ_1 . The location X gets two copies and is split, resulting in strategy σ_2 . The outgoing and incoming probabilities of both X s are identical. For both X s, the outgoing probabilities are preserved, but the incoming probabilities are halved between the two copies.

We now show how to construct such a strategy. For every bottom strongly connected component B , choose any $\tau_B \in T$ and $e_B \in B \times B$ such that $\text{Val}(\sigma \mid B) = \mathcal{D}(e_B, \tau_B \mid \sigma)$ and $\sigma(e_B) > 0$. Now, fix an observation $(c_1, \dots, c_n, c_{n+1})$. Let i be the smallest index such that (c_i, c_{i+1}) is equal to some e_B . If no index has this property or $i < n$, then the Attacker waits. Otherwise (c_n, c_{n+1}) is equal to a unique e_B and the Attacker attacks τ_B .

Let $c \in C$ be a state where the Defender starts patrolling. The proof builds on two basic properties of discrete-time Markov chains. No matter the starting state, the Defender will eventually visit some bottom strongly connected component B . Here, the Defender will eventually visit e_B with probability 1 and the Attacker attacks τ_B . Since M is chosen as the minimum over $\text{Val}(\sigma \mid B)$, no matter which bottom strongly connected component the Defender visits, the Attacker will achieve value at least M , which concludes the proof. $\square$

B Strategy Expansion

In [24, 12, 25], the parameters of the strategy are randomly initialized at the start of the optimization. However, it is also possible to warm-start from parameters provided by the user. This may be useful because we add memory only to some locations. Hence, it is possible that in the new strategy, some of the locations where we did not add memory could have the same optimal probabilities. By warm-starting, we would preserve these probabilities, saving time in optimization.

For that reason, we describe how to create a new strategy σ_2 from σ_1 given the profile function resulting from ADJUSTMEMORY. For notation purposes, let $\text{copies}(c) = |\text{profiles}(c)|$. The idea is to expand the strategy σ_1 such that every c has now exactly $\text{copies}(c)$ copies of c that preserves the value of the strategy.

If $\text{copies}(c) = 1$ for all but one state X which has $\text{copies}(X) = 2$, the procedure is simple. We create an identical copy X' of the state X . That is, we add a new state with the same outgoing probabilities as X . Then, every incoming edge to X gets halved between X and X' . In Figure 9, we show how this works on the positional strategy from the description of our method.

In general, let C' denote the new set of states induced by the new memory assignment mem_2 . For every state $c \in C$, let $c_1, \dots, c_{\text{copies}(c)}$ denote the corresponding new states in C' . Each c_i has the same distribution over C' equal to

$$\sigma_2(c_i, x_j) = \frac{\sigma_1(c, x)}{\text{copies}(x)}. \quad (9)$$

See Figure 10 for a visual example in the general case.

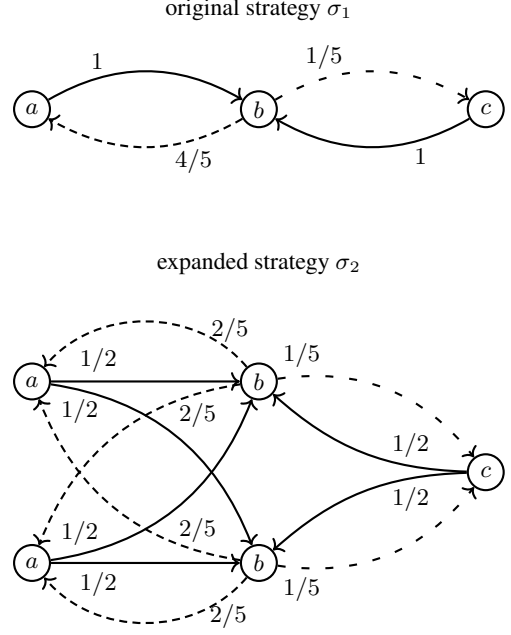


Figure 10. Demonstration of strategy expansion. The original strategy σ_1 is memoryless. Locations a and b both get two copies, resulting in expanded strategy σ_2 . The outgoing probabilities from c and a to b are halved between the two b s (from probability 1 to $1/2$). The outgoing probability from b to a is halved between the two a s (from $4/5$ to $2/5$ to each copy).

C Bounded Number of States

Assume we have an upper bound $L \geq |V|$ on the number of states for each σ_i . Let the current memory assignment be mem_1 and σ . We run ADJUSTMEMORY and collect the resulting mem_2 and attack profiles. If $\sum_v \text{mem}_2(v) \leq L$, we simply output mem_2 . Otherwise, for each attack profile, we sum the total value of the attacks with the given profile. That is, for each state c and an attack profile $\gamma \in \text{profile}(c)$, we define the *value* of γ as

$$\text{val}(\gamma, c) = \sum_{\substack{(e, \tau) \in \mathcal{E} \\ \text{sign}(\nabla_{x(c)} \mathcal{D}(e, \tau \mid \sigma)) = \gamma}} \mathcal{D}(e, \tau \mid \sigma).$$

Then, the idea is to take L profiles with the highest value, ensuring every state has at least one profile. The memory is then calculated in the same way as in the unbounded version.

Formally, let $\text{profiles}'(c)$ be the set of profiles of c , excluding one with maximal value (it does not matter which one). This is valid since every state must have a profile with an attack of maximal value. Then we flatten all the attack profiles into a single set $\mathcal{P} = \bigcup_c \{(\gamma, c) \mid \gamma \in \text{profiles}'(c)\}$. After that, we take the $L - |C|$ highest items from $\mathcal{P}$ where the order is given first by val and secondly by the order on the states (if more profiles of the same state have the same value, the order does not matter). Let us denote by $\#(c)$ the number of occurrences of a state c among the $L - |C|$ maximal items of $\mathcal{P}$. Finally, the new memory assignment mem_2 is

$$\text{mem}_2(v) = \sum_{1 \leq i \leq \text{mem}_1(v)} 1 + \#(v, i).$$

For strategy expansion, the value $1 + \#(c)$ corresponds to $\text{copies}(c)$.

D Complexity Analysis

Here, we describe the complexity analysis of our method, its bounded variation and strategy expansion.

Algorithm 1 The time complexity of Algorithm 1 is in

$$\mathcal{O}(f + |\mathcal{E}|g + |\mathcal{E}|^2|C|^2)$$

where f is such that the values of all attacks are calculated in $\mathcal{O}(f)$ and g is such that for each attack, the gradients with respect to the parameters are evaluated in $\mathcal{O}(g)$. Next, $|E|$ denotes the number of state edges such that its traversal probability is positive, and finally, $|\mathcal{E}|$ denotes the number of attacks taken into account.

Proof. In the algorithm, we keep a list of profiles for every state. First, we need to determine the values of all attacks in $\mathcal{O}(f)$. For every attack in the threshold, we calculate the gradients in $\mathcal{O}(g)$. Then, for each state, we calculate the sign of the vector in $\mathcal{O}(|C|)$ and decide if it's already in the list. This gives at most $|\mathcal{E}|$ iterations with equality of profiles being in $\mathcal{O}(|C|)$. Determining the size of the profiles and summing over vertices is done in $\mathcal{O}(C)$. $\square$

In practice, the evaluation for hard-constrained and blinded was faster than for linear. This is primarily because the first algorithm's core procedure is implemented in C++ with a dynamic program and uses a sparse representation of the strategy, unlike the second, which is in PyTorch and uses full matrices. In both cases, the actual bottleneck was in calculating the gradients.

Strategy Expansion We claim that strategy expansion is in $\mathcal{O}(|C|^2)$, where C is the new number of states.

Proof. For each new state c , evaluating the new outgoing probabilities can be easily done in $\mathcal{O}(|C|)$. This consists of copying the old probabilities divided by the number of new copies. This must be done for each state, hence in total $\mathcal{O}(|C|^2)$ operations. $\square$

Bounded case Let $M = \sum_v \text{mem}_2(v)$. If $M \leq L$, then the algorithm is the same as the normal case. Otherwise, we claim that in runs in

$$\mathcal{O}(|C||\mathcal{E}| + (M - |C|)\log(L - |C|))$$

Proof. Evaluating the value of all profiles of a single state can be implemented in $\mathcal{O}(|\mathcal{E}|)$. Hence, evaluating the value of all profiles of all states can be done in $\mathcal{O}(|C||\mathcal{E}|)$. The size of $\mathcal{P}$ is exactly $M - |C|$. Using heaps, extracting $L - |C|$ highest items of $\mathcal{P}$ takes $(M - |C|)\log(L - |C|)$. $\square$

E Experiments

Here, we offer the full report and description of our experiments. In the code appended, we further describe how to recreate our experiments. Moreover, we provide CSV files with the results of each experiment and the code used to generate the plots.

E.1 Setup

We ran our experiments on a server built on the Asus RS720A-E11-RS24U platform in the following configuration:

- Two 64-core AMD EPYC 7713 2.0 GHz processors (128 physical cores and 256 threads in total);
- 2 TiB DDR4 RAM 3200 MHz;
- 10 Gbps Ethernet connection;

- 2 SATA SSDs with 960 GB capacity in RAID 1;
- 2 6 TB NVMe drives in RAID 1;
- Red Hat Enterprise Linux operating system.

Every experiment was run with lowered priority by command `nice -n 10` and with a limit of 500GB on usable RAM by `systemd` scope.

The optimization process can be thought of as a finite sequence of steps, which we call a *run*. In each step, we perform forward and backward passes, updating the parameters. We executed 200 different runs for every patrolling graph and memory (fixed memory assignment or our method). The parameters were initialized randomly, and each run had a timeout of 180 seconds. In our method, we had bounded the number of states by 300.

To explain the setup, we divide the run into *epochs*. An epoch is the maximal subsequence where the memory assignment stays constant. For all baseline memory assignments, the whole run was a single epoch. There were usually multiple epochs in the case of runs with our method.

E.1.1 Terminating the optimization process

If a single epoch took more than 2000 steps, the whole run was terminated. The run was also terminated if the value of the best strategy found was below 10^{-9} or if the run hit a plateau.

A *plateau* occurs if, for n steps, the value has not improved by the relative threshold of ε . If the value of the best strategy found is v and after n steps, the best value is above $(1 - \varepsilon)v$, the run ends. We use values $n = 100$ and $\varepsilon = 10^{-5}$. Plateau detection is taken into account only after 500 steps of the current epoch.

E.1.2 Changing the epochs

Now, we describe how our method decides when to end an epoch and start a new one. Each epoch was given at least 200 steps. After that, we use plateau detection but for $n = 20$ (threshold is the same). If a plateau occurs during the epoch and the current epoch has improved the value by at least five percent, we change the memory according to the method, thus starting a new epoch.

Since both tools use Adam optimizer and noise, we naturally include that. Starting a new epoch includes resetting the optimizer with the new parameters and resetting the noise.

E.1.3 Hyperparameters of the tools

The tools for solving the particular models also include various hyperparameters. We use the parameters provided by the tools. More details can be found in the project configuration files. We consistently use the same hyperparameters across all experiments.

E.2 Offices

We evaluate our method on a benchmark introduced in [24] that also appeared in [12]. The goal is to patrol offices in an office building. The goal is to patrol offices (squares). The corridors (circles) connect the offices on the same floor. The floors are connected by staircases, which take 10 units of time to traverse. The traversal time between corridors is 2, and the time between offices and corridors is 5, as it involves opening the door and searching the office.

Here, all the targets (offices) are hard-constrained, with an attack length precisely equal to the shortest cycle visiting the entire building. The offices have equal value. The building is parametrized by

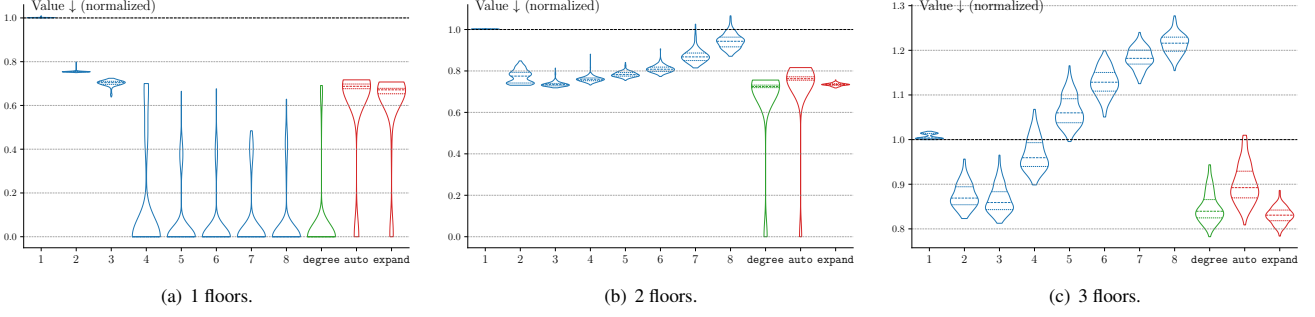


Figure 11. Statistics on the **Offices** benchmark with varying numbers of floors.

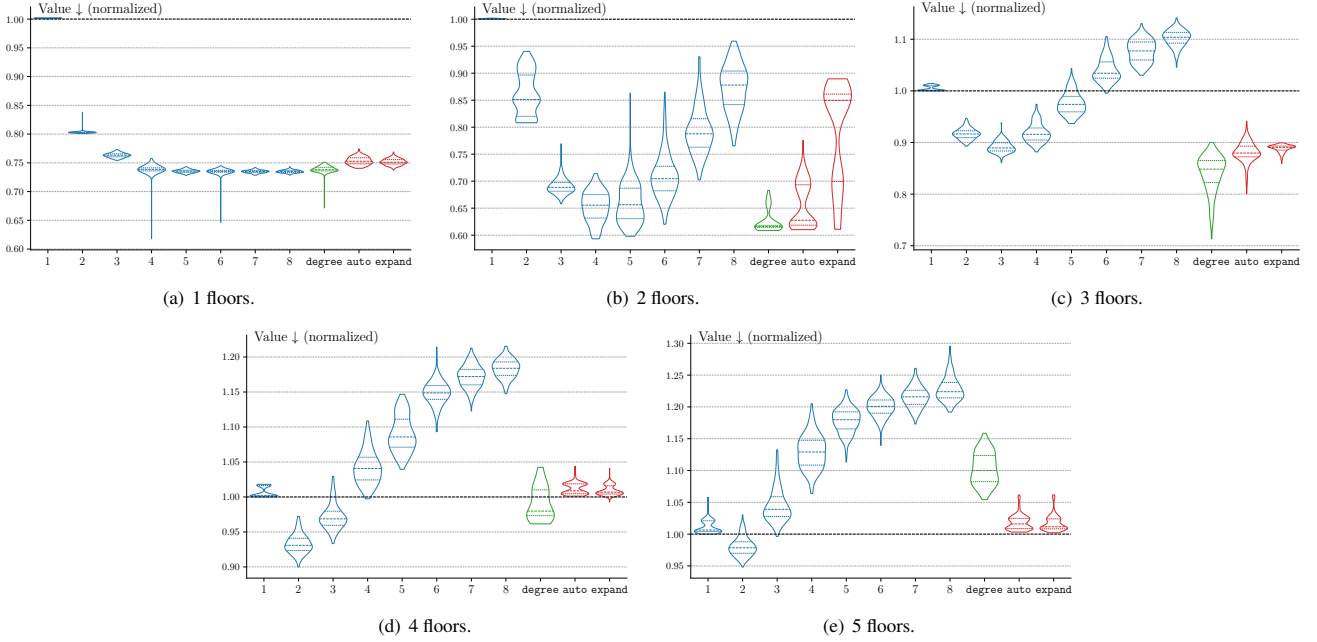


Figure 12. Statistics on the **Building** benchmark with varying number of floors.

the number of floors, and each floor has 4 corridors that join the offices. For short, we use n -floor to denote a building with n floors. Uniform memory assignment is used where every location is given m memory values and m ranges from 1 to 8. We ran the experiments for buildings with one, two, and three floors. In the original paper of [24], only a 1-floor building is evaluated.

E.3 Building

We continue with the second benchmark on the office building topology, in which the targets are blinded. This setting comes from [24]. The probability of discovering an ongoing attack upon each visit is 0.9. For buildings with n floors, the attack lengths of all targets are equal to $100n$. This is less than what is needed for the deterministic cycle, and the ratio to this length gets smaller with increasing n . We ran the experiments for a number of floors running from 1 to 5.

E.4 Airports

We evaluate our method on a model with linear targets. This model has so far been studied only in [25], where the authors use the fol-

lowing benchmark.

The goal is to patrol gates at an airport. The airport consists of three terminals and n halls. Each hall is connected to two gates, and the airport forms a tree. As a baseline, the authors use a strategy that cycles around the graph, visiting each target once on the cycle. The value of such a strategy is then the length of this cycle ($3n + 1$ for the airport with n halls) multiplied by the maximal value of gates. The authors use the memory of the baseline strategy to run the optimization tool. We denote this memory assignment as *degree*, as for each location, the memory is its degree in the graph. To demonstrate the need for memory, we also report results with memoryless assignment $m = 1$.

In the setting used by the authors, all gates have the same unit value. In that case, it is experimentally verified that the optimal strategy is actually the deterministic baseline. Following the authors, we generate 10 different airports, where the number of halls is a sequence

3, 4, 5, 7, 9, 12, 15, 19, 25, 30.

We evaluated the methods in two different target settings. One with

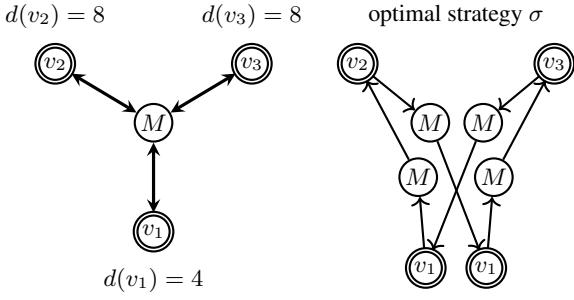


Figure 13. Graph for **Stars** benchmark and $k = 2$ (left). The strategy σ achieves perfect protection since v_1 is always visited every 4 steps, and both v_2 and v_3 are visited once on the cycle of length 8. Strategy σ_2 needs 4 states in M and 2 states in v_1 .

the uniform values of gates and the second where each value is randomly independently drawn from interval $[1, 10]$.

E.5 Stars

Here, we introduce a benchmark **Stars** where the goal is to patrol the leaves of a star. Let us denote by S_k a graph in a topology of a star with k leaves and internal location M . The benchmark for a graph with group size k is on a graph S_{k+1} , where $d(v_1) = 4$ and $d(v_i) = 4k$ for $2 \leq i \leq k + 1$. The defender needs to perform a cycle of length $4k$ of the form

$$v_1 \rightarrow M \rightarrow v_2 \rightarrow M \rightarrow v_1 \rightarrow \dots \rightarrow v_1 \rightarrow M \rightarrow v_k \rightarrow M \rightarrow v_1.$$

This requires $\text{mem}(v_1) \geq k$ and $\text{mem}(M) \geq 2k$. The other targets need just one memory value. We ran the experiments for k ranging from 1 to 5.

The case for $k = 2$ is drawn in Figure 13.

E.6 Terrains

One use of patrolling games is in guarding large terrains (see Related Work). Large terrains can be modeled as connected planar graphs, where each vertex represents a physical position in the terrain, and the length of edges is the time it takes to move between any two positions.

We compared the methods on a dataset of randomly generated connected planar graphs. The construction for a graph on n targets is as follows:

1. Generate n points uniformly in a unit 2D-box.
2. Find the Delaunay triangulation D .
3. Find a minimum spanning tree T of D where the length of edges is the Euclidean distance between the endpoints.
4. Add each edge from D to T with probability $1/2$.
5. Assign value to each target independently and uniformly from $[1, 10]$.

We generated 20 different graph for n ranging from 3 to 41 by steps of 2.

As with the previous experiments on airports, we choose the strategy that cycles around the graph in the shortest possible time as a baseline. Here, we use an approximate solution to the Traveling salesman problem. For technical details on graph generation and the baseline, see the appended code.

E.7 Effect of Strategy Expansion

From our experiments, we conclude that warm-starting the optimization process with strategy expansion is not significantly better. A notable exception is **Stars** for 4 groups. Here, **expand** leads to the optimal value of 0 as the only memory assignment. The reason for this is due to an interesting phenomenon where the value stagnates and suddenly drifts away to the global minimum. We observed this with constant memory assignment and on different graphs as well. In the runs with strategy expansion, we give this phenomenon more steps to appear. There are 20 steps of patience before the plateau; then, we expand the strategy, and a new epoch begins. Then, even if the value improves below the threshold and another begins, we are still virtually around the same local minima, albeit in bigger state space. This gives us at least 500 additional steps before terminating the run. Since the graph is small, they are executed before timeout.

However, in general, finding the extreme values is more probable without warm-starting. In cases where the memory needed is low, strategy expansion can lead to better convergence; this can be seen in **Terrains**.

F Example of a Run with More Iterations

Here, we show how the tool, together with our method, finds the optimal strategy in more iterations.

The example is on the **Stars** dataset for $k = 2$, i.e., the graph from Figure 13. We describe an actual run of the algorithm.

On the first iteration ($\text{mem}_1 \equiv 0$), the tool outputs a strategy σ_1 , drawn in Figure 17 (a). The value of σ_1 is 0.464. The maximal attacks contain attacks on all v_i when the Defender moves to some v_j , $i \neq j$. Since v_1, v_2 , and v_3 are deterministic, they have only one profile (0). The location M has three profiles $(-1, 1, 1)$, $(1, -1, 1)$, $(1, 1, -1)$ for each v_i since improving the probability automatically decreases probability to others. Hence the new memory is $\text{mem}_2(v_i) = 1$ and $\text{mem}_2(M) = 3$.

On the second iteration, the tool outputs a strategy σ_2 , drawn in Figure 17 (b). The value of σ_2 is 0.2705. The vertices v_2 and v_3 are still deterministic, so they have only one profile. All the states of M have three profiles; they have zero profiles for the attacks that occur as the Defender leaves these states and two profiles for strengthening the attacks on their two outgoing edges since they go to two different targets. The vertex v_1 has two profiles, one zero profile for attacks to v_1 , the second for covering the attacks on v_2 against v_3 . Hence the new memory is $\text{mem}_2(v_1) = 2$, $\text{mem}_2(v_2) = \text{mem}_2(v_3) = 1$ and $\text{mem}_2(M) = 9$.

On the third iteration, the tool outputs the optimal strategy with a value of 0.

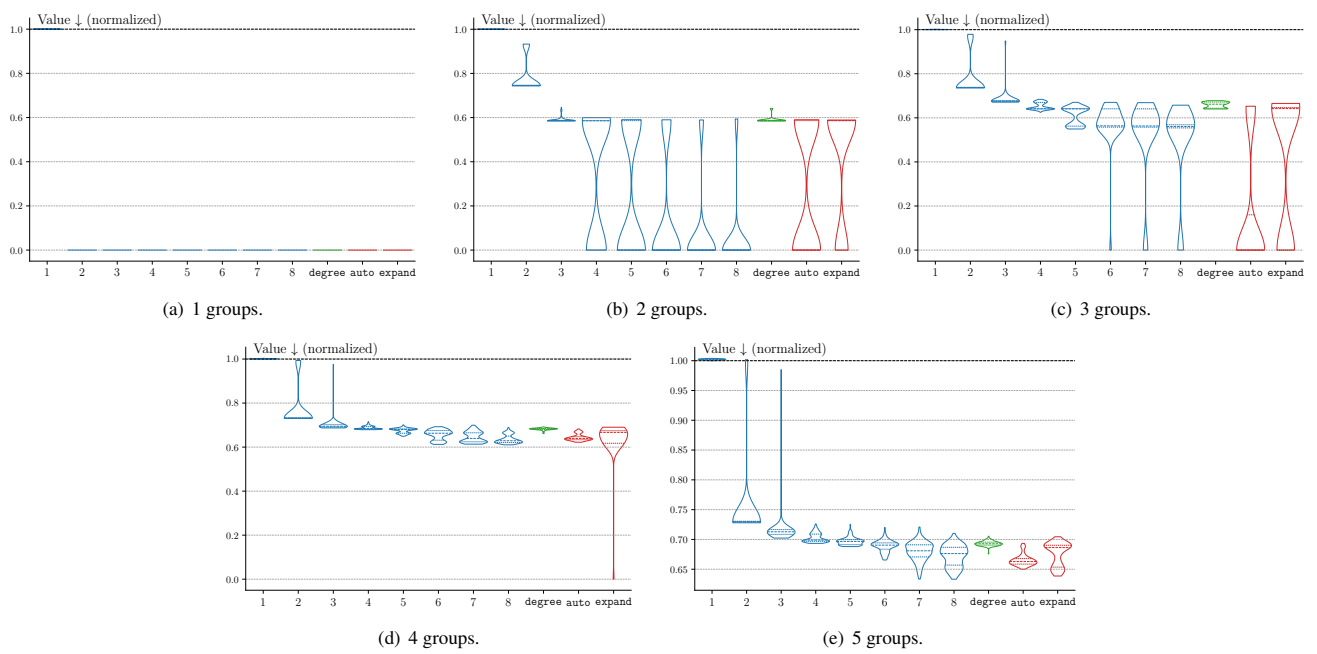
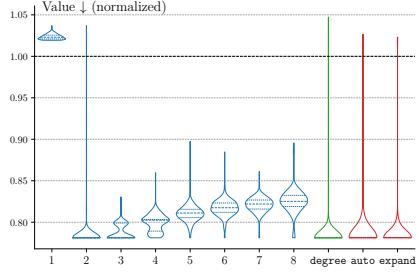
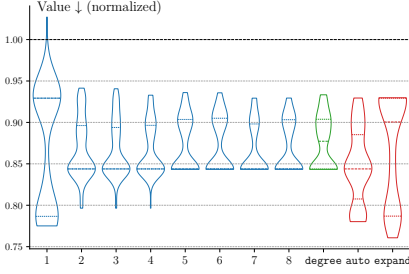


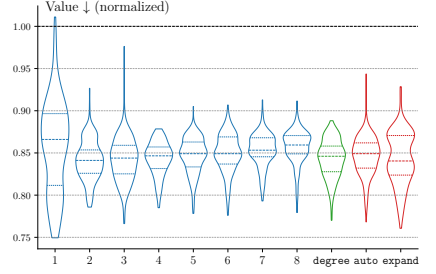
Figure 14. Comparison of the methods on the **Stars** benchmark with varying numbers of groups.



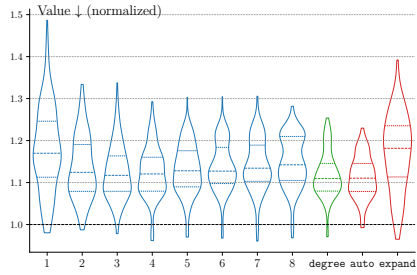
(a) Size 3.



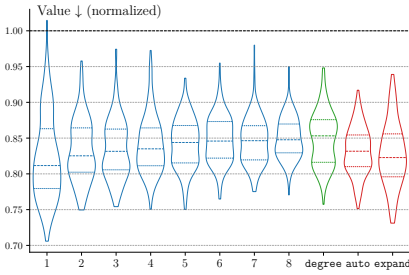
(b) Size 5.



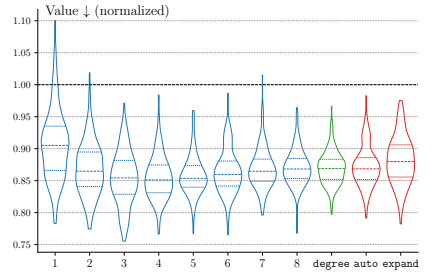
(c) Size 7.



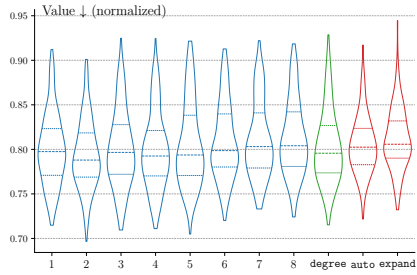
(d) Size 9.



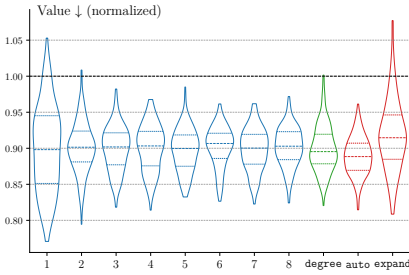
(e) Size 11.



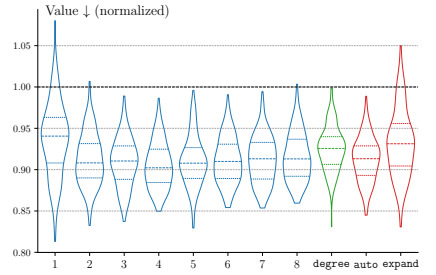
(f) Size 13.



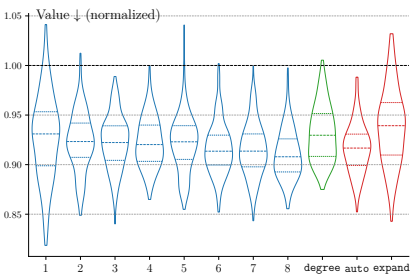
(g) Size 15.



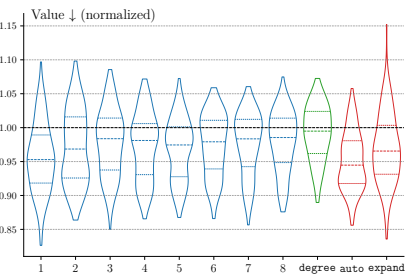
(h) Size 17.



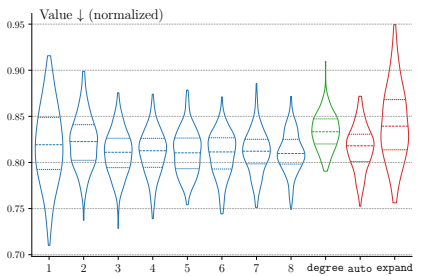
(i) Size 19.



(j) Size 21.



(k) Size 23.



(l) Size 25.

Figure 15. Comparison of the method on the **Terrains** benchmark for varying sizes.

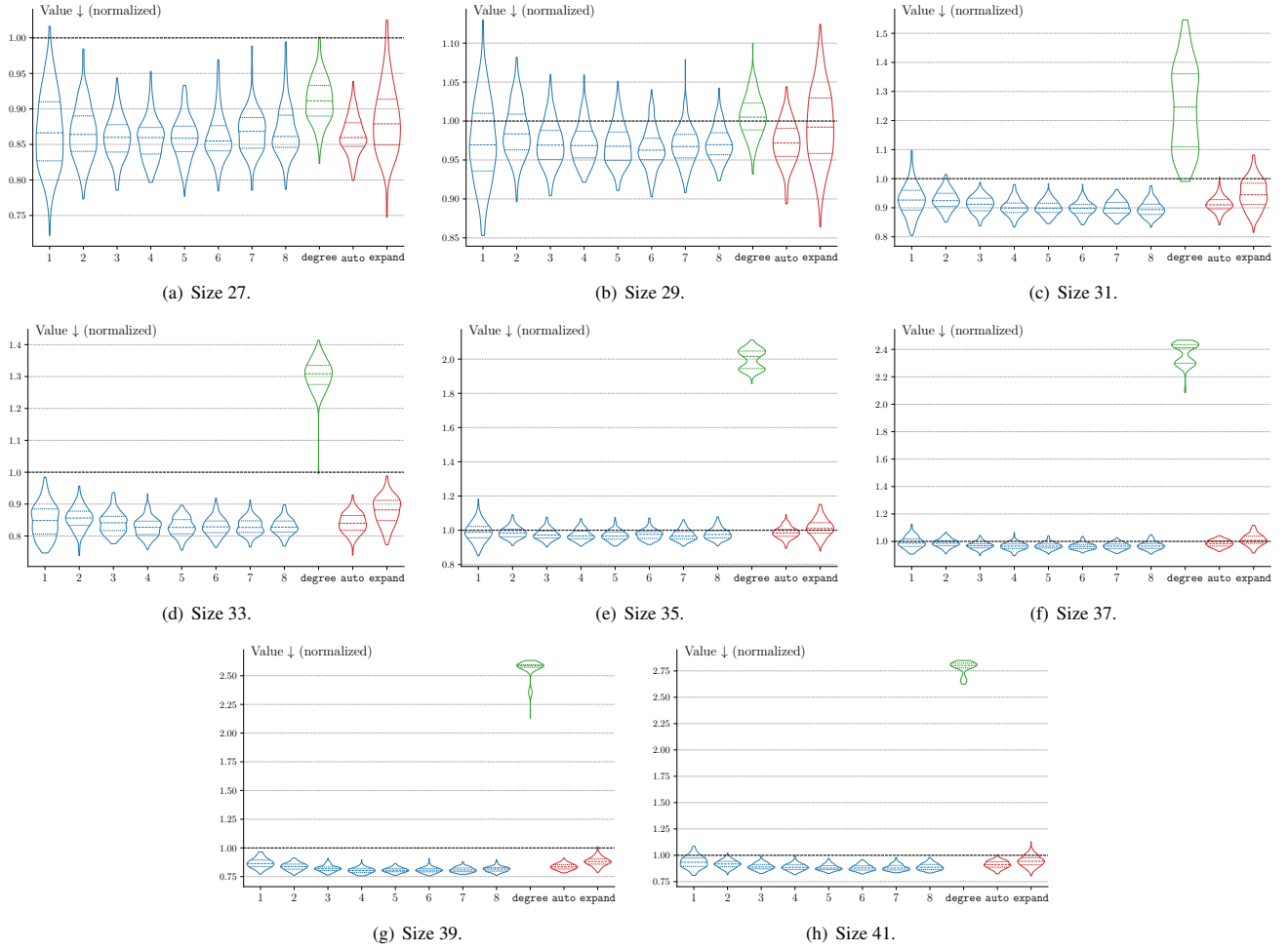


Figure 16. Comparison of the method on the **Terrains** benchmark for varying sizes (continuation).

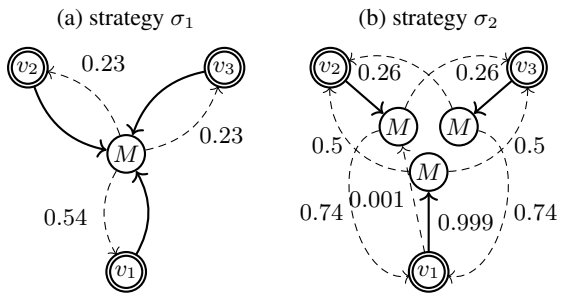


Figure 17. Example of a run of our algorithm. On the first iteration, the result is the strategy σ_1 (a). On the second iteration, the result is the strategy σ_2 (b). The probabilities are rounded for visual purposes.