Pursuit-Evasion on Trees by Robot Teams

Andreas Kolling, *Student Member, IEEE*, and Stefano Carpin, *Member, IEEE*

***Abstract*—We present Graph-Clear: a novel pursuit-evasion problem on graphs which models the detection of intruders in complex indoor environments by robot teams. The environment is represented by a graph, and a robot team can execute sweep and block actions on vertices and edges, respectively. A sweep action detects intruders in a vertex and represents the capability of the robot team to detect intruders in the region associated to the vertex. Similarly, a block action prevents intruders from crossing an edge and represents the capability to detect intruders as they move between regions. Both actions may require multiple robots to be executed. A strategy is a sequence of block and sweep actions to detect all intruders. When instances of Graph-Clear are being solved, the goal is to determine optimal strategies, i.e., strategies that use the least number of robots. We prove that for the general case of graphs, the problem of computation of optimal strategies is NP-hard. Next, for the special case of trees, we provide a polynomial-time algorithm. The algorithm ensures that throughout the execution of the strategy, all cleared vertices form a connected subtree, and we show that it produces optimal strategies.**

***Index Terms*—Distributed robot systems, pursuit-evasion games, surveillance systems.**

I. Introduction

THIS PAPER presents Graph-Clear: a pursuit-evasion problem on graphs suitable to model the detection of intruders in an environment by robot teams with limited-sensing capabilities. In Graph-Clear, an environment is represented by a weighted graph on which one can execute *sweep* actions on vertices and *block* actions on edges. A sweep action detects all intruders in a vertex, while a block action detects intruders that attempt to cross an edge. In Graph-Clear, it is assumed that all edges incident to a vertex are blocked while the sweep operation is executed. These actions represent routines that the robot team can execute in the actual environment. Because of the limited-sensing hypothesis, more than a single robot is, in general, necessary to successfully perform these intruder-detection operations. Weights on vertices and edges therefore associate a cost to each action, thus they denote the number of robots needed to execute it. The goal of Graph-Clear is to find a sequence of these actions, i.e., a so-called strategy, that detects all intruders in the environment using the least number of robots. Intruders are assumed to be omniscient and capable to move at unbounded speed. In particular, they are assumed to have full knowledge of the environment, of the pursuers' positions, and even of their strategy. We represent the possibility of an intruder being located somewhere with the concept of contamination. Initially, the entire graph is contaminated and each sweep and block clears contamination from vertices and edges. The task of finding any intruder is equivalent to removal of all contamination. Since intruders have full knowledge, recontamination of previously-clear vertices or edges occurs whenever an intruder has a path to that vertex or edge on which it cannot be detected.

To apply Graph-Clear and use strategies for the coordination of a real robot team, one needs to solve two subproblems. First, one has to provide implementations of sweep and block actions. These can differ widely and depend on the type of environment, robot platform, and sensors. Hence, they often require adherence to a particular sensing model or hardware. For a vertex, the corresponding implementation for the sweep action has to guarantee the detection of any intruder inside the region to which the vertex is associated, given that no intruder can enter or leave the region while sweeping takes place. Similarly, an implementation of a block action on an edge has to guarantee that no intruder can cross it undetected. The second subproblem is the automatic extraction of graphs from a given environment. This is not strictly necessary since graphs can be generated manually, but it is highly desirable for most applications, and it opens further research questions of interest. Our former results on extraction of graphs and weights from occupancy grid maps and implementations for sweep and blocking routines have been presented elsewhere [1] and are briefly discussed in Section III-A.

There is a rich literature about a variety of pursuit-evasion problems on graphs, which is often referred to as graph-searching. Graph-searching and its variations also require solutions to the above subproblems if one aims to utilize them to coordinate robot teams. The edge-searching problem [2] is perhaps the most prominent and oldest of these problems, and it is the most closely related to the model we present in this paper. To motivate the introduction of Graph-Clear, we therefore briefly describe its differences from edge-searching and its weighted variant. The edge-searching problem asks to determine a sequence of moves that detect all intruders in a graph using the least number of robots. A move consists of either placement or removal of a robot on a vertex, or sliding it along an edge. A vertex is considered guarded as long as it has at least one robot on it, and any intruder located therein or which attempts to pass through will be detected. A sliding move detects any intruder on an edge. In the weighted variant, weights on vertices denote the number of robots needed for each vertex to be considered guarded, while weights on edges denote the number of robots needed for a slide move to detect all targets [3]. Consequently, for each move in a sequence, one needs to additionally specify how many robots are used for it.

The key differences between weighted edge-searching and Graph-Clear are in the requirements imposed for the implementation of basic operations. To apply edge-searching, one needs

Manuscript received April 8, 2009; revised July 29, 2009. First published December 4, 2009; current version published February 9, 2010. This paper was recommended for publication by Associate Editor C. Stachniss and Editor L. Parker upon evaluation of the reviewers' comments.

The authors are with the School of Engineering, University of California, Merced, CA 95343 USA (e-mail: akolling@ucmerced.edu; scarpin@ucmerced.edu).

Digital Object Identifier 10.1109/TRO.2009.2035737

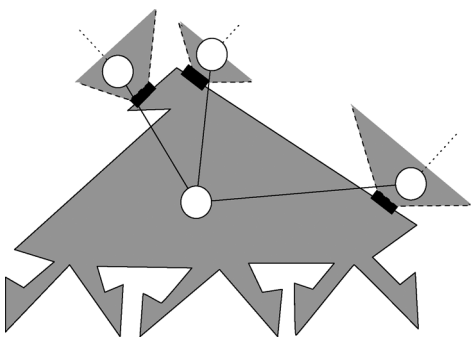


Fig. 1. Example of how a graph for Graph-Clear can relate to an actual environment. The environment is shown in gray with its graph embedded. All weights in this example are equal to one. Connections between regions that are connected by edges are shown in black. The center region is the "eagle" example redrawn from [4]. It can be cleared using the algorithm from [4] with only one robot and a simple gap sensor with sufficiently large range. During its execution, it recontaminates the top part of the region and, hence, cannot guarantee that no target enters the vertex undetected. We, hence, need blocks on the edges, i.e., to position sensors on the black regions. Note that the entire environment can be very large so that the sensor only satisfies the large range assumption within a vertex.

to provide implementations for guarding and sliding, while in Graph-Clear, one needs to implement sweeping and blocking. An implementation of guarding has to guarantee that all intruders in the associated region for the vertex are detected and, furthermore, that no intruder can enter or exit the region undetected. Sweeping does not require the latter. Instead, in Graph-Clear, while sweeping a vertex, we require a block on each edge to prevent targets that enter or exit. The consequence is that some robot algorithms cannot be used for guard operations. The example in Fig. 1 uses an algorithm for detection of targets inside the region of a vertex that does not satisfy the guarding requirements and, hence, is not directly suitable for edge-searching. To satisfy the guarding requirements, one would have to augment the algorithm by additionally using robots at the entrances. Then, the cost of this combined routine becomes a weight in weighted edge-searching, which represents the number of the robots needed to search the region and to keep entrances covered. But, once the robots searched the region, and hence, cleared the vertex, we still have to guard the vertex to prevent recontamination of its neighbors. In practice, this continued guarding after the actual search does not need to involve any of the robots that performed the search but only those that cover the entrances. However, in weighted-edge searching, we still pay the full cost for the guarding operation. This is because in edge-searching, guarding of a vertex performs two basic functions, namely, the prevention of spread of contamination from and to its neighbors and the detection of all intruders in the vertex. One could try to overcome this problem by having weights on edges represent the cost to enter a vertex and search and cover the entrances while the weight on the vertex only represents the cost to cover the entrances, but then, sliding along an edge costs more than guarding the vertex. Not only is this unintuitive, but the formulation of weighted edge-searching from [3] does not allow edge weights larger than the weight of the adjacent vertex. Even if this was remedied, there is yet another problem. Suppose we clear the center vertex and then one of its neighbors. At this point, the entrance to the neighboring vertex does not need to be blocked any further and the weight for guarding the center vertex should change to reflect this. Since weights are fixed, this cannot be captured. There is no immediate remedy for this, since in edge-searching, the spreading of contamination is avoided only by actions on vertices and never on edges. In Graph-Clear, on the other hand, the search of a region and the prevention of recontamination from neighboring regions are separated, and the latter occurs on edges. In colloquial terms, in edge-searching, the intruder movement is restricted by robots in vertices, while in Graph-Clear, we move this capability to the edges, effectively disentangling detection in a vertex from the prevention of recontamination. Evidently, this is useful for vertices that represent complex regions and edges that are simple connections between these regions. On the other hand, edge-searching is useful for simple vertices and complex connections between these, such as the network of tunnels example often mentioned as a motivation for edge-searching. Another important distinction between weighted edge-searching and Graph-Clear is discussed in detail after the definitions for Graph-Clear in Section III-A. It relates to a counterintuitive consequence of the addition of weights to the traditional edge-searching problem. Allowing simultaneous moves can improve solutions to the weighted edge-searching problem.

The main motivation for Graph-Clear is the design of robot algorithms that will ultimately run on large robot teams. Yet, this paper is primarily devoted to the formalization and analysis of Graph-Clear as a formal pursuit-evasion problem on a graph. This formalization allows us to address the main computational challenges resulting from the consideration of large environments and large robot teams in a formal and deterministic setting. Other challenges, such as probabilistic scenarios with faulty sensors and imprecise actuators, implementations of sweep and block actions on real robots, and algorithmic extraction of graphs from robot maps are addressed in separate papers [1], [5] and are the subject of further work. More precisely, this paper presents the following four original contributions.

1) Graph-Clear is rigorously formalized. This formalization allows us to exploit a number of formerly developed literature results in related areas (see Section III).
2) We prove that the decision version associated with the Graph-Clear problem is NP-hard. This result was announced in one of our former papers, but the proof has never been published (see Section IV).
3) For the special case of contiguous strategies which ensure that all intruder-free vertices are always connected, we prove that recontamination does not help (see Section VI).
4) Given that for the general case of graphs, the problem is NP-hard, we focus our attention on trees. In Section V, we start presenting some terminology useful for clearing trees, and then, in Section VII, we present an algorithm to produce contiguous strategies for trees. The algorithm is shown to use the least number of robots and has time-complexity quadratic in the number of vertices.

To ease reading, all proofs are given in the Appendix, with the exception of the proof of NP-hardness.

II. Related Work

Surveillance tasks with robots have been investigated in manifold variations. They differ with respect to the considered environments, communication and mobility constraints, sensor models, and the number of robots used. Our review focuses on selected publications from visibility-based pursuit-evasion, graph-searching, and cooperative surveillance.

Visibility-based pursuit-evasion is one of the more prominent research areas with sound theoretical results. The field was pioneered by Suzuki and Yamashita [6] with the introduction of the k-searcher, which emits k beams to detect intruders, and the ∞-searcher, which is a point source for such beams. Sufficient and necessary conditions for existence of a search schedule with a k-searcher in simple polygons were presented as well. Yamashita *et al.* [7] introduced tight upper and lower bounds on the so-called search number of a polygon. Some of the upper bounds were derived from graph-searching, discussed later in this section. A first complete algorithm to solve the visibility-based pursuit-evasion problem was given by LaValle *et al.* [8] for the ∞-searcher. The approach is motivated by information states that change when critical boundaries are crossed by a searcher. The information states are associated to gap edges of the sensors, i.e., all edges of the polygon covered with the sensor that are adjacent to free space. These critical boundaries partition the polygon into cells. Combining this decomposition of the polygon in graph form with the information state produces a new exponential-size graph in which a solution is computed. Guibas *et al.* [9] established NP-hardness for detection of the minimal number of ∞-searchers needed for any polygon through a reduction of some instances of visibility-based pursuit-evasion to instances of graph-searching. Furthermore, they showed that there exist polygonal environments that a single pursuer can clear but only if $\Omega(n)$ recontaminations are allowed, where n is the number of edges of the polygon. This result applies to unlimited-range sensors. In Graph-Clear, we will show that recontamination does not help to improve clearing strategies. It is currently unclear whether recontamination helps in all robotic scenarios, in particular, with very limited sensing when the sensing range is smaller than the smallest distance between any two distinct obstacles. Finally, Park *et al.* in [10] presented a quadratic algorithm for solving the visibility-based pursuit-evasion problem for one pursuer. Necessary and sufficient conditions for searchability are given, as well as a proof of the conjecture by Suzuki and Yamashita stating that a polygon searchable by an ∞-searcher is also searchable by a 2-searcher. Curved environments were first considered in [11]. The approach therein extends the critical boundaries from [8] to smooth boundaries of the environment based on inflections and bitangents. The environment is a simply-connected one and the pursuer has omnidirectional vision. For polygonal environments, LaValle *et al.* [12] presented an algorithm for a pursuer with only a flashlight, i.e., a 1-searcher. The algorithm solves the problem by Suzuki and Yamashita for 1-searchability and produces a search strategy if one exists. Simple polygons with n edges and m concave regions are considered, which leads to an algorithm with complexity $O(m^2 + m \log n + n)$. The basis of the algorithm is a so-called *visibility obstruction diagram*, which is a 3-partition of the configuration space. In this diagram, certain paths, called winning paths, lead to a strategy of the pursuer in the polygon. The search space of the diagram is reduced to a skeleton by considering critical points on the boundary of the polygon. In this structure, a path can be found efficiently. Another variant using sentries (immobile sensing devices that can be placed by a robot) is investigated by Guilamo *et al.* in [13]. The authors develop an online algorithm for unknown simply-connected environments for simple-gap sensors. The motion strategy is based on critical events regarding the gaps detected by the sensor, i.e., appearances, disappearances, and splits and merges of gaps. The approach uses sentries that the pursuer can place and recollect. The number of dropped sentries is bounded by $O(\log m)$, where m is the number of bitangents related to critical events. One main result is that if one robot can clear the environment, then so can the robot using a gap-navigation tree with at most two sentries. Numerous methods presented therein make use of results on gap navigation from [14] and [15]. Sachs *et al.* in [4] presented an online algorithm for a point pursuer moving in an unknown, simply-connected, piecewise-smooth planar environment. The pursuer is only equipped with a sensor that measures depth discontinuities. Also, the controls are minimalist, as only wall-following or a movement along the measured depth discontinuities is allowed. Furthermore, imperfect control is assumed. The approach incrementally builds a navigation graph based on the motion primitives. The information state, i.e., possible locations for the intruder, is superimposed on this graph forming the so-called information graph. An online version is obtained by computing preliminary solutions in the information graph. The algorithm is complete and enables the limited pursuer to clear the same environments that a pursuer with a map, perfect localization, and perfect control can clear. Results with multiple robots, i.e., two 1-searchers, are due to Simov *et al.* in [16]. The environment is therein restricted to a simple polygon. The authors presented an $O(n^2 + nm^2 + m^4)$ complete algorithm to compute a search strategy in a polygon with n edges and m concave regions. It is also based on an information state graph using an elaborate geometrical characterization of the polygon. Yet another visibility-based variant considering bounded speed is investigated in [17]. The environment is a simply-connected polygon and the algorithm incorporates the notion of time by considering reachability sets, i.e., contaminated areas grow with time and form these sets instead of filling the accessible space instantaneously. Recently, Yu and LaValle [18] presented a method that infers possible target locations from combinatorial sensor data from multiple robots. Space is separated into visible and multiple-connected shadow regions and target location is inferred as shadow regions merge and split. Robot movement, however, is not controlled by the algorithm.

Another closely-related area of research is known as graph-searching. In the following, we can present only a small selection of the vast literature. A recent survey is available in [19]. The origins of graph-searching relate back to [2], in which Parson *et al.* proposed the first pursuit-evasion problem on graphs now known as edge-search. The problem consists of an initially

contaminated graph G in which searchers traverse edges to capture an arbitrarily fast intruder. Contamination is used to represent the possibility of the intruder occupying an edge. Intruders can move across all vertices that are not guarded by searchers. The problem is to find the smallest number of searchers, known as the *search number* $s(G)$, with which one can guarantee to capture the intruder. Megiddo *et al.* in [20] showed that the decision variant of finding $s(G)$ is NP-complete. The proof is a reduction of the well-known *min-cut into equal-sized subsets* problem [21]. The authors also presented a linear-time algorithm to find the search number in trees and an $O(n \log n)$ algorithm to compute a corresponding strategy. The proof relies on an early manuscript from 1982, which was later published as [22], in which LaPaugh showed that for a pebbling version of edge-search, recontamination is not necessary for optimal strategies. Another proof that recontamination is not necessary in edge-searching is given by Bienstock and Seymour in [23]. The original edge-search problem has been explored in many other variants such as node search [24], mixed search, inert search, and domination search [25]. Many of these variants have been connected to graph-layout problems. The search number is shown to be the cut-width of G if the maximum degree of any vertex is 3 [26]. Relations between the search number and vertex separation are established in [27] and methods to compute the vertex-separation layout problem were shown to be applicable to compute strategies in $O(n \log n)$ and determine the search number in $O(n)$ for certain instances of the problem. Improved vertex-separation layout methods from [28] can find strategies in $O(n)$. The node-search number was shown in [24] to be equal to the vertex separation plus one. Most relevant for our purposes is a graph-searching variant from [3] in which Barriere *et al.* considered weighted graphs and introduced the concept of contiguous strategies, thus requiring that all cleared vertices form a connected subgraph. They presented an algorithm that computes contiguous strategies in linear time on trees, which is based on computing labels on edges. It is important to note that edge-searching and Graph-Clear differ significantly, i.e., in edge-searching, searchers move across edges, thereby clearing them, and guard vertices, which restricts the intruder movement. In Graph-Clear, edges are blocked (the equivalent of guarding), and vertices are cleared.

In cooperative multirobot research, there is a third strain of investigation concerned with cooperative surveillance of moving targets with large robot teams. Most of this research focuses on heuristics and often simplifies the environment significantly. Theoretical results are rare in this area, but the methods cope with inferior sensing capabilities and larger team sizes. A popular problem is the cooperative multirobot observations of multiple moving targets (CMOMMT) defined by Parker [29]. Solutions to CMOMMT include those given in [29] and [30], the latter including proven bounds for performance. More complicated environments are considered in [31] and heuristics for distributing the robots across parts of the environments are developed. For the probabilistic detection of intruders, there are heuristics developed by Moors *et al.* in [32]. The authors create a graph by random placement of vertices in the environment until all parts are covered, assuming that a vertex covers the area a sensor covers at the same location. Edges connect all vertices with overlap in coverage, and an A* search computes the plan for robots on this graph.

III. Problem Formulation

A formal definition of Graph-Clear is presented in this section. It is assumed that the reader is familiar with graphs and trees, and is referred to [33] for the basic notation and terminology that we use. The first part presents a formulation in terms of graph theory concepts, and the language is chosen accordingly. The connection between Graph-Clear and real-world problems is presented in Section III-A.

Definition 1 (Surveillance Graph): A *surveillance graph* is a triple $G = (V, E, w)$, where (V, E) is an undirected graph with vertex set V, edge set E, and $w : V \cup E \to \mathbb{N}^+$ as a weight function.[1] Vertices and edges in a surveillance graph have a state. The state of a vertex can be *clear*, or *contaminated*, while the state of an edge can be *clear*, *contaminated*, or *blocked*. If x is a vertex or an edge, its state is indicated as $\nu(x)$.

Notation: Depending on the context, edges will be indicated either as e or as (v_i, v_j), with $v_i, v_j \in V$. Throughout the paper, the notation (u, w) indicates an undirected edge between vertices u and w. If v is a vertex, $edges(v)$ is the set of all edges having v as end point. The degree of a vertex v is the number of edges having v as end point, i.e., $degree(v) = |edges(v)|$. If G is a graph, $V(G)$ is its set of vertices and $E(G)$ the set of edges. Also, possible state values will be abbreviated using the letters $\mathcal{R}$ for clear, $\mathcal{C}$ for contaminated, and $\mathcal{B}$ for blocked.

Assumption: From here onwards, unless otherwise stated, we shall assume that $|V| = n$ and that $|E| = m$.

Definition 2 (State Space and State of a Surveillance Graph): The *state space* of a surveillance graph G is the set

$$\mathcal{V}(G) = \{\mathcal{R}, \mathcal{C}\}^n \times \{\mathcal{R}, \mathcal{C}, \mathcal{B}\}^m.$$

The *state* of the surveillance graph G is an element $\nu = \{\nu_1, \ldots \nu_{n+m}\} \in \mathcal{V}(G)$ such that $\nu_i = \nu(v_i)$ for $i = 1 \ldots n$, and $\nu_{n+j} = \nu(e_j)$ for $j = 1 \ldots m$.

The state of a graph is a string of symbols from the alphabet $\{\mathcal{R}, \mathcal{C}, \mathcal{B}\}$ indicating the state of every vertex and edge. The first n symbols indicate the state of vertices and the last m the state of edges.

Definition 3 (Recontamination Path): Let G be a surveillance graph with state ν, and let $x, y \in V \cup E$. A *recontamination path* between x and y is a path of edges and vertices on which no edge is blocked, i.e., for all edges e_i of the path, we have $\nu(e_i) \neq \mathcal{B}$.

Note that while defining the concept of recontamination path, we generalize the usual definition of path in a graph. Rather than defining paths only between a couple of vertices, we also consider paths between edges and between a vertex or an edge, or *vice versa*. In every situation, we impose that the edges along the path should not be blocked. Two types of actions can be applied to a surveillance graph, namely, *blocking* on edges and

[1] In this paper, $\mathbb{N}^+$ indicates the set of positive natural numbers, while $\mathbb{N}$ indicates the set of natural numbers (i.e., including 0).

sweeping on vertices. The goal of these actions is to change the state of vertices and edges so that no contaminated edges or vertices remain. While blocking can be applied to any edge, sweeping can be applied to a vertex v only if all edges originating from v are blocked.

Definition 4 (Action Set and Actions): The *action set* of a surveillance graph G is the subset of $\{0,1\}^{n+m}$, where each element $a = \{a_1, \ldots, a_{n+m}\}$ (called *action*) satisfies the following constraint.

1) If $a_i = 1$ with $1 \le i \le n$, then $a_{n+j} = 1$ for each edge $e_j \in edges(v_i)$.

If $a_i = 1$ with $1 \le i \le n$, we say that the action a sweeps vertex v_i, and if $a_{n+j} = 1$ with $1 \le j \le m$, we say that action a blocks edge e_j. The action set of G is indicated as $\mathcal{A}(G)$.

Definition 5 (Sweeping and Blocking Cost): Let G be a surveillance graph. The *sweeping cost* of a vertex $v \in V$ is $w(v)$, while the *blocking cost* of an edge $e \in E$ is $w(e)$.

Definition 6 (Cost of an Action): Let G be a surveillance graph, and let $a \in \mathcal{A}(G)$ be an action. The *cost of action* a is

$$c(a) = \sum_{i=1}^{n} a_i w(v_i) + \sum_{j=1}^{m} a_{n+j} w(e_j).$$

The former definition states that the cost of a is the sum of the blocking and sweeping costs for the edges blocked and vertices swept by a. It follows that the cost of sweeping a single vertex v is as follows:

$$s(v) = w(v) + \sum_{e_j \in edges(v)} w(e_j) \tag{1}$$

because in order to sweep a vertex, it is necessary to block all its edges.

Definition 7 (Transition Function): Let G, $\mathcal{V}(G)$, and $\mathcal{A}(G)$ be defined as earlier. The *transition function* ζ maps a state and an action into a new state

$$\zeta : \mathcal{V}(G) \times \mathcal{A}(G) \to \mathcal{V}(G).$$

Given $a \in \mathcal{A}(G)$ and $\nu \in \mathcal{V}(G)$, the new state ν' is defined as follows.

1) If $a_{n+j} = 1, 1 \le j \le m$, then $\nu'(e_j) = \mathcal{B}$.
2) If $a_i = 1, 1 \le i \le n$, then $\nu'(v_i) = \mathcal{R}$.
3) If $\nu_{n+j} = \mathcal{B}, a_{n+j} = 0, 1 \le j \le m$, and no recontamination path between e_j and $x \in V \cup E$ with $\nu(x) = \mathcal{C}$ exists, then $\nu'_{n+j} = \mathcal{R}$.
4) If there exists a recontamination path between $x \in V \cup E$ and $y \in V \cup E$ with $\nu(y) = \mathcal{C}$, then $\nu'(x) = \mathcal{C}$.
5) $\nu'_i = \nu_i$ otherwise.

The transition function establishes how the state of G changes when an action is applied. The five cases can be described in words as follows:

1) edges where a block action is applied become blocked;
2) vertices where a sweep action is applied become clear;
3) blocked edges where a block action is not applied anymore become clear if there is no recontamination path involving them;
4) vertices or edges for which a recontamination path toward a contaminated vertex or edge exists become contaminated;
5) vertices or edges maintain their previous state if none of the former cases apply.

Definition 8 (Strategy): Given a graph state $\nu \in \mathcal{V}(G)$, a *strategy* $\mathcal{S}$ for ν is a sequence of actions $a_1, a_2, \ldots, a_k$ that when applied in sequence clears all elements in G, i.e,

$$\underbrace{\zeta \ldots \zeta(\zeta}_{k \text{ times}}(\nu, a_1), a_2) \ldots, a_k) = \underbrace{\{\mathcal{R}, \mathcal{R}, \ldots, \mathcal{R}\}}_{m+n \text{ times}}.$$

Definition 9 (Cost of a Strategy): Let $\mathcal{S} = \{a_1, \ldots, a_k\}$ be a strategy. The *cost of strategy* $\mathcal{S}$ is

$$ag(\mathcal{S}) = \max_{i=1\ldots k} c(a_i). \tag{2}$$

We now can formally define the Graph-Clear problem.

Definition 10 (Graph-Clear Problem): Let G be a surveillance graph whose state is $\nu = \{\mathcal{C}, \mathcal{C}, \ldots, \mathcal{C}\}$. Determine a strategy $\mathcal{S}$ for ν of minimal cost.

From now onward, before a strategy is applied to a surveillance graph, we assume that the state of all its elements is $\mathcal{C}$, unless stated otherwise. We distinguish between two types of strategies using the concept of contiguity adapted from [3].

Definition 11 (Contiguous and Noncontiguous Strategies): Let G be a surveillance graph in state $\nu = \{\mathcal{C}, \mathcal{C}, \ldots, \mathcal{C}\}$. A strategy $\mathcal{S}$ for G is contiguous if the subset of cleared vertices and cleared or blocked edges always forms a connected subgraph of G. Otherwise, a strategy is said to be noncontiguous.

This distinction is useful, since contiguous strategies on trees turn out to be easier to compute, as presented in Section VII. Another type of strategy is the following.

Definition 12 (Progressive Strategy): A *progressive strategy* is a strategy for a surveillance graph G in state $\nu = \{\mathcal{C}, \mathcal{C}, \ldots, \mathcal{C}\}$ preventing recontamination.

In particular, we will later on focus on progressive contiguous strategies.

Definition 13 (Cost of a Graph): Let G be a surveillance graph, and let $\mathcal{S}$ be a strategy of minimal cost for G. We define the *cost of graph* G as $ag(G) = ag(\mathcal{S})$.

Similarly, for a graph G in any state ν, we write $ag(G, \nu) = ag(\mathcal{S})$, where $\mathcal{S}$ is a strategy with minimal cost for G in state ν. For a subset of vertices $U \subseteq V$, let $a_1, \ldots, a_k$ be a sequence of actions that clears all vertices of U starting from state ν with minimal cost $\max_{i=1\ldots k} c(a_i)$.[2] We write $ag(U, \nu)$ for this minimal cost.

The cost of a strategy is the number of robots needed to clear the environment according to the sequence of actions defined by the strategy. Since we seek strategies with the fewest robots, we will consider only strategies for which at most one vertex at a time is swept. This approach is justified by the following lemma whose simple proof is omitted.

Lemma 1: Let $\mathcal{S} = \{a_1, \ldots, a_k\}$ be a strategy for G. Then, there exists a strategy $\mathcal{S}'$ with cost $ag(\mathcal{S}') \le ag(\mathcal{S})$ that sweeps no more than a vertex at a time.

[2]Such sequences are not necessarily strategies, unless $U = V$, but they can be thought of as partial strategies.

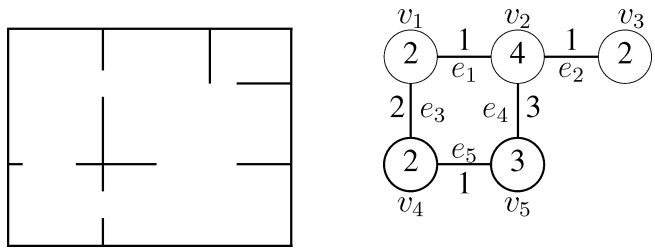


Fig. 2. Example environment and one possibly associated surveillance graph. Numbers inside vertices are the sweeping costs, and numbers on the edges are blocking costs.

$\nu(G)$	a	$c(a)$
$\mathcal{CCCCC}$ $\mathcal{CCCCC}$	10000 10100	5
$\mathcal{RCCCC}$ $\mathcal{BCBCC}$	00010 10101	6
$\mathcal{RCCRC}$ $\mathcal{BCBCB}$	01100 11011	12
$\mathcal{RRRRC}$ $\mathcal{BBRBB}$	00001 00011	7
$\mathcal{RRRRR}$ $\mathcal{RRRBB}$	00000 00000	0
$\mathcal{RRRRR}$ $\mathcal{RRRRR}$		

Fig. 3. Possible strategy to solve the Graph-Clear problem associated with the graph shown in Fig. 2. The first column displays the status, the second the applied action, and the third the cost. Note that in the third row, an action sweeping two vertices at the same time is applied and that a final action removing all blocks is executed in the end (with 0 cost). The cost of this strategy is 12, i.e., the maximum value read in the third column. It is easy to see that such a strategy is not optimal.

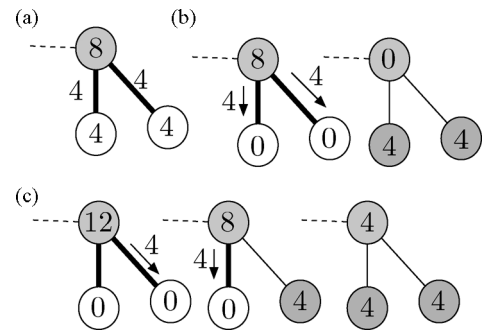


Fig. 4. Consequences of allowing simultaneous moves in weighted edge-searching. (a) Graph with its weights. (b) Graph with eight robots on the top vertex and none in the bottom vertices. The arrows indicate two sliding moves with four robots that finish clearing the graph with eight robots when executed simultaneously. (c) How to clear the graph with strictly sequential moves with the same recontamination rules but needing more robots.

A. An Example of Graph-Clear

We now work out a simple example outlining the connection between Graph-Clear and practical surveillance scenarios. A surveillance graph is used to model complex environments. For our illustration, we choose to intuitively associate vertices with rooms and edges with connections between rooms (i.e., doors or corridors). Fig. 2 shows a simple indoor environment and one possible surveillance graph. A contaminated vertex is a room that may hide an intruder, while a clear vertex is known to be intruder-free. Intruders may also hide in connections between rooms, and therefore, edges can also be clear or contaminated. Robots equipped with sensors are used to detect intruders. In our problem formulation, an intruder disappears as soon as it is detected by a robot (i.e., it falls within its sensing range). A blocking operation applied to an edge ensures no intruder can pass through that connection without being detected by the robots blocking it. Since our focus is on scenarios involving robots with limited sensing capabilities, more than one robot may be necessary to block large connections (see the values displayed in Fig. 2). To detect all possible intruders inside a room, a sweeping operation is performed. Once again, since robots have limited capabilities, multiple robots are requested to make sure one room is free of intruders. Since a room may have multiple entrances, it is necessary to block all of them to prevent intruders from entering parts of the room that have been swept already (recontamination). The recontamination path concept adds significant challenges and asymmetries to the problem. In fact, while we assume that robots are capable of detecting intruders only when they fall within a limited sensing range, our definition of recontamination implies that as soon as recontamination is possible, it immediately occurs. We therefore suppose intruders have complete knowledge of the environment and of the robots' positions, and they may move with unbounded speed on continuous paths to take instantaneous advantage of the existence of recontamination paths. The Graph-Clear problem asks how to schedule sweeping and blocking operations so that eventually, the environment is completely cleared using the least number of robots. Fig. 3 shows a possible strategy to solve the Graph-Clear problem on the environment depicted in Fig. 2.

Before Graph-Clear can be of practical relevance in a robotic scenario, it requires a method to partition an environment into regions, which then become vertices in the surveillance graph as well as implementations for the sweeping and blocking actions. A first solution extracting surveillance graphs from occupancy grid maps is presented in [1]. It is based on detecting narrow parts of the environment using its Voronoi Diagram. The method not only extracts the graph, but it also assigns appropriate weights based on given sensing abilities of the robots, as well as a predetermined clearing method for vertices. From now onward, we assume that for a given sensor model, the corresponding surveillance graph is available.

Before proceeding, we consider a fundamental difference between weighted edge-searching and Graph-Clear, apart from the motivation presented in Section I and Fig. 1. This difference results from the addition of weights. Recall that in edge-searching, a single move is either 1) moving along an edge, 2) placing a robot on a vertex, or 3) removing a robot from a vertex. In weighted edge-searching, these moves additionally receive an integer representing the number of robots participating in the move. The weight on the edge or vertex determines how many robots are needed so that the move clears the edge and guards the vertex. This seems like a straightforward extension of the previous model, but it has counterintuitive consequences. Consider the example in Fig. 4. Therein, allowing simultaneous moves can improve the number of robots needed. This results from the fact that the guarding operation on one vertex can need more than the sum of the slide and guarding operations toward all neighbors.

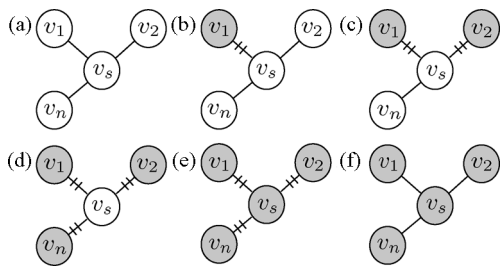


Fig. 5. Construction of an optimal strategy for a star. Cleared and contaminated vertices are gray and white, respectively. Blocked edges are marked with a double-stroked line. First, all leaves are cleared, leaving the edge to the leaf blocked. For leaf v_i, $i = 1, \ldots, n$, and the total cost while clearing it is $i + 1$. Finally, the center vertex is cleared with cost $n + 1$.

IV. Complexity of Graph-Clear

In one of our former papers, we stated a theorem claiming NP-hardness of the Graph-Clear problem [34], but the full proof was omitted due to space constraints. We offer the complete proof based on the new notation developed in Section III. The proof is mainly an adaption of the proof of NP-completeness of edge-search on a graph presented in [20]. The key idea is to substitute complete graphs used in [20] with stars defined in the following. The method constructs a surveillance graph with all weights equal to 1 and the statement hence also holds for the simpler case of unweighted surveillance graphs. Throughout this section, all weights are assumed to be equal to 1.

We first define the concept of *stars* and prove a bound on the clearing cost.

Definition 14 (Stars): A *star* of order n, written G_n, is a surveillance graph that is a tree with $n + 1$ vertices $v_0, \ldots, v_n$ and n leaves. The vertex v_s that is not a leaf shall be called *center*.

Lemma 2: Let G_n be a star of order n. Then, $ag(G_n) = n + 1$.

Proof: Let v_0 be the center of G_n. According to (1), the cost to clear v_0 is $n + 1$. To clear G_n with $n + 1$ agents, clear v_0 first and block all its n edges. Then, use the $n + 1$th robot to clear each leaf (see Fig. 5).

Let us consider the following decision version of the Graph-Clear problem:

INSTANCE: $G = (V, E, w)$, a surveillance graph with $w(x) = 1 \forall x \in V \cup E$, and a natural number P

QUESTION: is $ag(G) \leq P$?

Proving that this problem is NP-hard relies on a polynomial reduction to the *min-cut into equal-sized subset* problem (MCIESS from now on), a problem known to be NP-complete [21]. MCIESS is posed as follows.

INSTANCE: An undirected graph $G = (V, E)$ with an even number of vertices, and a natural number K.

QUESTION: Is there a partition of V into two subsets V_1 and V_2 with $|V_1| = |V_2| = \frac{1}{2}|V|$ such that $|\{(u, v) \in E : u \in V_1, v \in V_2\}| \leq K$?

Theorem 1: The decision version of Graph-Clear is NP-hard.

Proof: Let $G = (V, E)$ and $K > 0$ be an instance of the MCIESS problem. Let $n = |V|, d = \max_{v_i \in V}\{degree(v_i)\}$, $N = 6(d + K)$, and $M = nN(n + 2)$. An instance $H = (U, F, w), P$ of the Graph-Clear problem is built in polynomial time as follows:

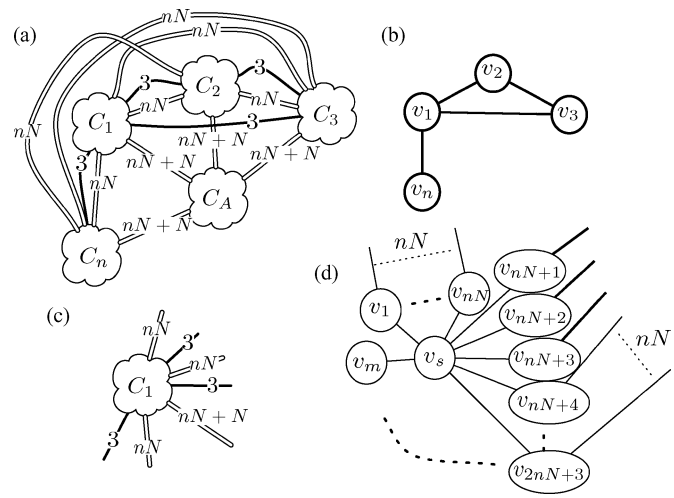


Fig. 6. Large graph constructed from an instance of the MCIESS. (a) Constructed surveillance graph from the MCIESS graph in (b). A star is represented by a cloud, a bundle of nN or more edges by a double line, and three edges by a thick line. (c) Closeup of the star C_1 and its edges to other star. (d) C_1 is shown in more detail with its center, connectors, and leaves.

1) for each $v_i \in V$, create a star of order M called C_i (i.e., $C_i = G_M$, with $i = 1 \ldots n$);
2) let C_A be an additional star of order M (i.e., $C_A = G_M$);
3) add edges between leaves of the star with at most one edge for each leaf:
   a) add nN edges for each pair $C_i, C_j, i \neq j$, note that $i, j \in \{1, \ldots, n, A\}$;
   b) add N more edges between each C_i and C_A;
   c) add three more edges between C_i and C_j if $(v_i, v_j) \in E$;
4) $w(x) = 1 \forall x \in U \cup F$;
5)

$$P = (M + 1) + \left(\frac{n}{2}\right)^2 nN + 3K.$$

Note that it is possible to give each leaf v of a C_i at most one edge, since we have M such leaves and never add more than $M - 1$ edges in total to any C_i. Fig. 6 visualizes the construction. All vertices that received an edge during the construction will be called *connectors*, and all that did not remain leaves. There is at least one leaf remaining for each C_i. We now show that the Graph-Clear instance H, P admits a positive answer if and only if the MCIESS instance G, K does.

Assume the instance G, K admits a positive answer, i.e., we have a partition of V into V_1 and V_2 s.t. $K' \leq K$ edges connect $V_1 = \{v_1, \ldots, v_{n/2}\}$ and $V_2 = \{v_{n/2+1}, \ldots, v_n\}$. Let us then consider the following strategy $\mathcal{S}$ for H: Clear $C_1, \ldots, C_{n/2}, C_A, C_{n/2+1}, \ldots, C_n$ in this order. Being more specific, according to Lemma 2, the cost to clear the leaves and center of C_1 is $M + 1$. The number of edges from C_1 to other C_is is at most $n^2N + N + 3d < M$, and hence, there is at least one vertex in C_1 with degree 1. We start clearing C_1 by clearing each degree-1 vertex. Then, we clear the center

and keep all its edges blocked. This procedure costs $M+1$. After having cleared the center, we remove all blocks from edges to leaves while retaining those to connectors, which are still contaminated. The number of remaining blocks is at most $n^2N+N+3d$. Consider a path between the clear center of C_1 and another C_i. This path is not a recontamination path because the edge to C_1 is blocked. An additional cost of 2 is incurred to move this block to the edge ending on the center of C_i. Doing this for all connectors of C_1 costs at most $n^2N+N+3d+2$. Hence, with cost $M+1$, we can clear C_1, and leave at most $n^2N+N+3d$ blocks in edges in other C_is, effectively reducing the cost needed to clear them.

Clearing C_2 now has cost no higher than $(M-nN)+1$, since it is equivalent to clearing a connected G_{M-nN} as nN edges to C_2's center have been cleared after clearing C_1. Additionally, we still have the cost of blocking edges from the first step. So, the total maximum cost is $(M-nN)+1+n^2N+N+3d$. After clearing C_2, the total number of blocks that have to remain is $2\times(n^2N+N+3d)-nN$, since we no longer have to block the nN edges between C_1 and C_2, but we have all those between C_2 and all contaminated C_is. Generalizing this formula, the cost of each step $2\le i\le n/2$ is

$$[M-(i-1)nN]+1+(i-1)(n^2N+N+3d-(i-2)nN) \tag{3}$$

which gives us at the worst step $n/2$

$$M+1+\left(\frac{n}{2}-1\right)\left(\frac{n}{2}+1\right)nN+\left(\frac{n}{2}-1\right)(N+3d)<P.$$

For C_A, we need at most

$$M+1+\left(\frac{n}{2}\right)^2nN+3K'\le P.$$

For C_i with $n/2+1\le i\le n$, an upper bound analog to (3) applies. Hence, there exists a strategy for H of cost at most P. By definition, this means $ag(H)\le P$, and therefore, the answer to the instance H,P is positive as well.

For the converse, suppose that $ag(H)\le P$. This means there exists a strategy for H of cost not higher than P. By Lemma 1, while clearing H using the optimal strategy, there has to be a step at which $n/2+1$ centers of the C_is are cleared and $n/2$ are not. Consider the step of clearing the $(n/2+1)$th center, and let C_j be the star to which it belongs. Let us assume that $C_j\ne C_A$. The least possible cost at this point is

$$(M+1)+\left(\frac{n}{2}\right)^2\cdot nN+\frac{n}{2}N.$$

This bound is derived as follows: $M+1$ is the cost to clear the center of $C_j(1)$, $(\frac{n}{2})^2nN$ is the cost to block paths between clear centers and contaminated centers different from C_j, and $\frac{n}{2}N$ is the cost to block paths between clear centers and C_A. However, $(M+1)+\left(\frac{n}{2}\right)^2\times nN+\frac{n}{2}N>P$, which is a contradiction since the strategy has cost not higher than P. Therefore, $C_j=C_A$. Clearing C_A costs $M+1$ and blocking the cleared centers from the contaminated centers costs at least $\left(\frac{n}{2}\right)^2nN$, resulting from the nN edges added in construction step 3a between the $\left(\frac{n}{2}\right)^2$ pairs of stars. The additional edges from step 3c between cleared and contaminated centers result from the original instance of MCIESS, but there can be at most $3K$ such edges between these centers since $P=(M+1)+\left(\frac{n}{2}\right)^2nN+3K$. Hence, there are at most K edges between vertices in the original MCIESS instance that correspond to cleared and contaminated stars. Hence, we can define $V_1=\{v_i: C_i \text{ has clear center}\}$ and $V_2=\{v_j: C_j \text{ has contaminated center}\}$ and get a cut between V_1 and V_2 with at most $\lfloor(3K+2)/3\rfloor\le K$edges. □

V. Label-Based Contiguous Strategies on Trees

The result presented in Section IV leaves little hope of being able to find optimal strategies for all instances of Graph-Clear with polynomial-time complexity. This stimulates research to study more constrained versions of the problem. In particular, we will show that if one restricts the attention to contiguous strategies on trees rather than graphs, then optimal solutions can be found with time-complexity polynomial in the number of vertices. Alternatively, one can seek for approximated solutions for graphs, along the spirit of the algorithms presented in [35]. Approximate algorithms for Graph-Clear are currently under investigation but will not pursued any further in this paper.

From now onward, we assume to operate on trees (i.e., connected acyclic graphs) and will therefore use the letter T rather than G. The problem of converting a surveillance graph into a tree is discussed in [34]. Furthermore, we will also impose the requirement that all strategies are contiguous. The first algorithm to compute contiguous strategies on trees, built upon previous work by Barriere *et al.* [3], is also presented in [34]. The algorithm does not always produce optimal strategies, thus it motivates the extension presented in Section VII. Nonetheless, we will shortly present it, because it introduces concepts needed also for the optimal algorithm presented later. In fact, the new algorithm presented in Section VII can be seen as a generalization of the one illustrated in this section.

For the sake of completeness, we mention there are also algorithms for computation of noncontiguous strategies on trees [36], [37], but they will not be discussed further in this paper as their performance is still hard to evaluate.

The algorithm that computes contiguous strategies from [34] is as follows. Numeric labels associated with edges are computed as described later, and then, a strategy is produced based on the labels' values. Let $T=(V,E,w)$ be a surveillance tree. For each edge (v_x,v_y), two labels λ_{v_x} and λ_{v_y} are defined as follows.

1) λ_{v_x} is the cost of clearing the contaminated subtree rooted in v_y after clearing v_x.
2) λ_{v_y} is the cost of clearing the contaminated subtree rooted in v_x after clearing v_y.

Labels are computed bottom-up starting from edges incident on leaves. Due to the symmetry in the definitions of λ_{v_x} and λ_{v_y}, we discuss only the computation of λ_{v_x}. Consider an edge $e=(v_x,v_y)$. If v_y is a leaf node, i.e., $degree(v_y)=1$, then

$$\lambda_{v_x}(e)=w(v_y)+w(e)=s(v_y)$$

since in order to clear v_y, it is necessary to block the only edge it has. Next, let us assume v_y is an internal node, i.e., degree$(v_y)>1$. Let us indicate the k neighbor vertices that

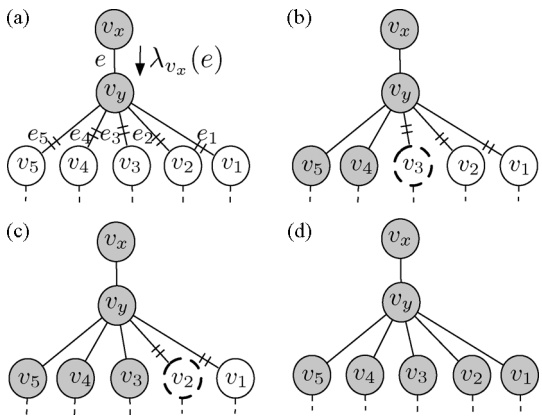


Fig. 7. Contiguous strategy on a tree is executed based on the labels on edges. Blocked edges are crossed through twice, and cleared vertices are gray. A vertex with dashed lines attached represents an entire subtree rooted at that vertex. A subtree being cleared is marked with the corresponding root vertex drawn in thick dashed lines. The label associated with this procedure is shown in (a) with the direction of the robots marked by an arrow.

are different from v_x as $v_1, \ldots, v_k$, $k = degree(v_y) - 1$. Let $e_i = (v_y, v_i)$, $i = 1 \ldots k$, and let us define

$$\rho_i = \lambda_{v_y}(e_i) - w(e_i). \tag{4}$$

Reorder vertices so that $\rho_i \geq \rho_{i+1}$. The subtree rooted at v_y will be cleared according to the following strategy. First, block all edges $e_1, \ldots, e_k$, and clear v_y. Then, fully clear the subtree rooted at v_k. After clearing the subtree rooted at v_k, no block on e_k is necessary anymore, and then, remove it. Next, clear the contaminated subtree rooted at v_{k-1}, and then, remove the block from e_{k-1}. Next, clear the subtree rooted at v_{k-2}, and so on. Accordingly, in this strategy, the total cost when clearing the contaminated subtree rooted at v_i is composed of all blocks at the other neighbors and the costs to clear the subtree itself, represented by the label $\lambda_{v_y}(e_i)$. This becomes

$$c(v_i) = \lambda_{v_y}(e_i) + \sum_{l=1}^{i-1} w(e_l). \tag{5}$$

The value for $\lambda_{v_x}(e)$ is then computed as follows:

$$\lambda_{v_x}(e) = \max\{s(v_y), \max_{i=1,\ldots,k}\{c(v_i)\}\}.$$

Fig. 7 illustrates this approach graphically. Having ordered all neighboring vertices so that $\rho_i \geq \rho_{i+1}$ ensures that λ_{v_x} is minimized.[3] Once all labels are computed, a strategy $\mathcal{S}_v$ that starts clearing the tree T from a vertex v is defined as follows. Let $v_1 \ldots v_k$ be all k vertices that are neighbors of v. First, block all edges to v and clear v. Then, recursively clear the contaminated subtree rooted at v_i, with i from k to 1, using strategy $\mathcal{S}_{v_i}$ while blocking all edges $e_1, \ldots, e_{i-1}$. The cost of $\mathcal{S}_v$ is the following:

$$ag(\mathcal{S}_v) = \max\left\{s(v), \max_{i=1,\ldots,k}\{c(v_i)\}\right\}. \tag{6}$$

[3]To show this, assume there was an optimal ordering s.t. $\rho_i < \rho_{i+1}$, and show that you can then swap v_i and v_{i+1}.

Once all labels have been computed, it is possible to find the vertex v for which the quantity defined in (6) is minimized. Such a vertex is the starting point to clear the tree.

Given a surveillance tree, labels λ_v can be computed in $O(n \log d)$, where n is the number of vertices and d the maximum degree across all vertices. The complete algorithm is presented in [34]. However, it is possible to produce specific instances of Graph-Clear where the depicted algorithm does not yield an optimal contiguous strategy. This limitation motivates the formalism and ideas presented in the next section.

VI. Existence of Optimal Contiguous Strategies That Are Progressive

This section shows that recontamination is not necessary for optimal contiguous strategies on trees. This result is essential for the construction of such strategies in polynomial time described in Section VII. The proof is based on the concept of *cuts* that we will introduce first.

A. Cuts

Definition 15 (Cut): Let T be a surveillance tree. A *cut* of T is a subset of $V(T)$ whose induced subgraph is connected. We will indicate cuts with the letter γ, and the cut $\gamma = V(T)$ is called *full cut*. Γ is the set of all cuts of T.

Cuts can be thought of as describing all cleared vertices of a tree T. Hence, it is useful to describe the cost of blocking its boundary so that recontamination does not occur.

Notation: Let G be a surveillance graph and $X \subset V(G)$. Then

$$\delta X = \{(x, y) \in E(T) | x \in X \wedge y \notin X\}.$$

δX is the subset of edges connecting vertices in X to vertices not in X.

Definition 16 (Cut Blocking Costs): Let γ be a cut of T. Its *cut blocking costs* is

$$b(\gamma) = \sum_{e \in \delta V(\gamma)} w(e). \tag{7}$$

In colloquial terms, b is the cost to prevent recontamination of γ once it is fully cleared. By definition, a cut γ can be cleared by the execution of a sequence of actions with cost $ag(\gamma, \nu)$, given that T is in state ν. As a shorthand, we say that we *execute* a cut γ at cost $ag(\gamma, \nu)$. Consequently, executing γ modifies the state of T. Let us formalize the notion of sequential execution of cuts and define its cost.[4]

Definition 17 (Cut Sequence and its Cost): Let Γ be the set of all cuts for a surveillance tree T, and let T be in state ν^1. We define a *cut sequence* $\mathcal{S}$ as a sequence of r cuts $\gamma^1, \ldots, \gamma^r$ from Γ, where γ^r is a full cut. At step $l = 1, \ldots, r$, cut γ^l is executed modifying the state ν^l to ν^{l+1}. The cost of $\mathcal{S}$ at step l is $c^l = ag(\gamma^l, \nu^l)$, and the cost of $\mathcal{S}$ is

$$c(\mathcal{S}) = \max_{1 \leq l \leq r}(c^l).$$

[4]In the sequel, when talking about multiple cuts, we will use the term *sequence* when the order matters, as opposed to sets.

If all cuts in a cut sequence $\mathcal{S}$ are executed, T is eventually cleared because the last cut is a full cut by definition. It is immediately seen that such a sequential execution of cuts leads to a strategy for T; hence, the use of the letter $\mathcal{S}$ for both strategies and cut sequences. The two different terms are introduced because they focus on different perspectives. A strategy is a sequence of actions and, hence, specifies exactly which sweep and block actions are taken at every step. In contrast, a step in a cut sequence only describes which vertices have to be in clear state, namely, those belonging to the cut. How vertices of a cut are cleared depends on the strategy that executes it and is not specified by the cut. For example, $\gamma^1 = V(T)$ is the simplest cut sequence, but executing γ^1 involves finding a strategy for all of T at once. On the other hand, cut sequences that add at most one vertex from one cut to the next can be immediately converted into a strategy.

B. Recontamination Does Not Help

Results presented in this section are similar to the work by Bienstock and Seymour [23] for edge-search, and their definitions are herein adapted for Graph-Clear. In particular, they introduced the concept of crusades that we borrow and call simple-cut sequence. An analogous crusade-based construction was also used in [3]. Although the basic ideas are similar, many mathematical technicalities are different, and details are therefore fully worked out in this paper.

Definition 18 (Simple-Cut Sequence): Let T be a surveillance tree. A *simple cut sequence* in T is a cut sequence $\mathcal{S} = \gamma^1, \ldots, \gamma^r$ such that $|\gamma^1| = 1$ and $|\gamma^i \setminus \gamma^{i-1}| \leq 1$ for all $2 \leq i \leq r$. For simple-cut sequences, with a slight abuse of notation we write v_i for $\gamma^i \setminus \gamma^{i-1} \neq \emptyset$. If γ^i is a cut in a simple-cut sequence, its *frontier* $f(\gamma^i)$ is defined as follows:

$$\begin{array}{ll} s(v_1), & \text{if } i = 1 \\ b(\gamma^i) + s(v_i) - \sum_{e \in \delta v_i \cap \delta \gamma^i} w(e), & \text{if } i > 1, \gamma^i \setminus \gamma^{i-1} \neq \emptyset \\ b(\gamma^i), & \text{if } i > 1, \gamma^i \setminus \gamma^{i-1} = \emptyset. \end{array} \tag{8}$$

The *frontier* of a simple-cut sequence $\mathcal{S}$ is

$$f(\mathcal{S}) = \max_{1 \leq i \leq r} \left\{ f(\gamma^i) \right\}. \tag{9}$$

The definition of simple-cut sequences allows steps where $|\gamma^i \setminus \gamma^{i-1}| = 0$. This situation may arise in the uninteresting case $\gamma^i = \gamma^{i-1}$ or when $|\gamma^{i-1} \setminus \gamma^i| > 0$. This latter case corresponds to recontamination of all vertices in $\gamma^{i-1} \setminus \gamma^i$. Let us now define progressiveness for cut sequences.

Definition 19 (Progressive Cut Sequence): A cut sequence is a *progressive cut sequence* if $\gamma^1 \subseteq \gamma^2 \subseteq \ldots \subseteq \gamma^r$.

Note that progressiveness of a cut sequence is conceptually different from progressiveness of strategies. A progressive cut sequence can be executed by a strategy that is not progressive [consider, for example, the progressive cut sequence $\gamma^1 = V(T)$]. Given a surveillance tree T, we will now show that an optimal contiguous strategy implies the existence of an optimal contiguous strategy that is also progressive. This is done in three steps by first considering simple-cut sequences, then cutting sequences that are both simple and progressive, and finally, connecting these to existence of an optimal contiguous strategy that is progressive. These three steps are formalized with the following claims.

Lemma 3: Let T be a surveillance tree, and let $\mathcal{S}_c$ be an optimal contiguous strategy for T of cost $ag(\mathcal{S}_c) \leq k$. Then, there exists a simple-cut sequence $\mathcal{S}$ for T such that $f(\mathcal{S}) \leq k$.

Proof: See the Appendix.

Next, it is possible to show that for any simple-cut sequence of bounded frontier, there is a simple progressive cut sequence whose frontier is not greater.

Theorem 2: If there exists a simple-cut sequence $\mathcal{S}$ in T with $f(\mathcal{S}) \leq k$, then there exists a simple progressive cut sequence in T with frontier not greater than k.

Proof: See the Appendix.

Finally, by connecting simple progressive cut sequences to strategies, we can show the main result of this section.

Theorem 3: Let T be a surveillance tree, and let $\mathcal{S}_c$ be an optimal contiguous strategy for T of cost $ag(\mathcal{S}_c)$. Then, there exists a progressive contiguous strategy of cost $ag(\mathcal{S}_c)$.

Proof: See the Appendix.

The main message of this section is that it is possible to construct optimal contiguous strategies on trees even when imposing that no recontamination should occur. It should be noted that the same can be shown to hold for graphs but with a slightly more complicated proof of Theorem 2. Such a proof for graphs and including noncontiguous strategies would turn the NP-hardness proof of Section IV into an NP-completeness proof, since strategies without recontamination are in NP. For all practical purposes, the result on trees suffices since the algorithm in the next section is restricted to trees.

VII. Optimal Contiguous Strategies: A Polynomial Algorithm

Given that we established the existence of at least one optimal contiguous strategy that is progressive, our goal is now to devise an algorithm that computes one efficiently. To do so, we first introduce a final class of cut sequences, called *full-cut sequences*, and identify some notable properties. Based on these findings, we next develop an $O(n^2)$ algorithm to compute an optimal contiguous strategy.

A. Full-Cut Sequences

Let T be a fully-contaminated surveillance tree and let v_y be the first vertex cleared by an optimal contiguous progressive strategy. Since v_y will never be recontaminated, we can consider the subtrees at each neighbor $v_1, \ldots, v_k$ of v_y separately. More precisely, for each $i = 1, \ldots, k$, we will write T_i for the subtree of T rooted at v_y with all edges of v_y removed except the edge to v_i (see Fig. 8). Each of the T_is can be thought of as a surveillance tree with the same weights induced by the w function defined on T and its own state. Given that a cut sequence alters the state of T, we will indicate with ν_i^l the state of T_i before γ^l is executed on T.

We want to construct an optimal-cut sequence $\mathcal{S}_{v_y}$, which starts clearing v_y first, and then removes all contamination from T. Its first cut is $\gamma^1 = \{v_y\}$, which we can execute with cost

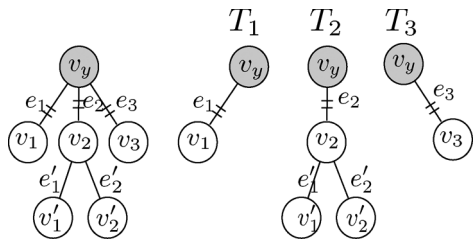


Fig. 8. Given v_y we define subtrees T_i as seen in the figure.

$ag(\gamma^1, \nu^1)$ leading to a new state ν^2. Note that, according to the notation introduced earlier, ν^2 induces a state ν_i^2 for each T_i. The goal is now to find optimal-cut sequences for each T_i starting at state ν_i^2 that we can use to continue $\mathcal{S}_{v_y}$ and turn it into an optimal-cut sequence for the whole tree T. The building blocks for these sequences are identified by the following definition.

Definition 20 (Full-Cut Sequence): Let T be a surveillance tree in state $\nu^1 = \{\mathcal{C}, \ldots, \mathcal{C}\}$, $v_y \in V[T]$ and Γ_{v_y} the set of all cuts containing v_y. A *full-cut sequence for* v_y, indicated as $\bar{\mathcal{S}}$, is built as follows. Sort all cuts of Γ_{v_y} s.t. $ag(\gamma, \nu^1)$ is increasing. All cuts with identical $ag(\gamma, \nu^1)$ are additionally sorted with $b(\gamma)$ increasing. First, add $\gamma^1 = \{v_y\}$ to $\bar{\mathcal{S}}$. Next, add the first cut of Γ_{v_y} with $b(\gamma) < b(\gamma^1)$. Then, add every next cut γ from Γ_{v_y} with the next larger $ag(\gamma, \nu^2)$ such that $b(\gamma)$ is smaller than for the previously added cut. Repeat this process until the full cut is added to $\bar{\mathcal{S}}$.

Observe that since, by definition, the full cut has $b(\gamma) = 0$, this definition is well posed for every T and terminates after a finite number of steps. Note that we did not require that a full-cut sequence be either progressive or simple. A full-cut sequence $\bar{\mathcal{S}}$ for v_y has a useful property that $ag(\gamma^l, \nu^l) = ag(\gamma^l, \nu^2)$ for all $2 \leq l \leq r$. This is formalized by the following more general lemma.

Lemma 4: Let $\mathcal{S} = \gamma^1, \ldots, \gamma^r$ be a cut sequence such that $ag(\gamma^l, \nu^l)$ is monotonically increasing for all indexes l within the range $b \leq l \leq r$. [5] Then, for all $b \leq l \leq r$

$$ag(\gamma^l, \nu^b) = ag(\gamma^l, \nu^l).$$

Proof: See the Appendix.

Next, for each T_i, let $\bar{\mathcal{S}}_i$ be a full-cut sequence for v_y. It is worth observing that, by definition, $v_y \in T_i$ for each T_i, so a full-cut sequence for v_y in T_i is well defined. For each i, let us indicate cuts in $\bar{\mathcal{S}}_i$ as γ_i^j. To each cut $\gamma_i^j \in \bar{\mathcal{S}}_i$ with $j \geq 2$, i.e., excluding the first cut $\{v_y\}$, associate a value ρ_i^j defined as follows:

$$\rho_i^j = ag(\gamma_i^j, \nu_i^j) - b(\gamma_i^{j-1}), \quad j \geq 2. \tag{10}$$

Notice the similarity between (10) and (4), and, in fact, the latter generalizes the former. Let

$$\bar{\Gamma} = \bigcup_{i=1,\ldots,k} \bar{\mathcal{S}}_i \setminus \{v_y\}$$

[5]This is the case for full-cut sequences picking $b = 2$. For $b = 1$, the lemma does not necessarily hold for full-cut sequences since the cost for executing $\gamma^1 = \{v_y\}$ is allowed to be greater than subsequent costs.

and order them with ρ increasing. This ordering is consistent with the ordering of each subsequence by construction of full-cut sequences. Notice that $\gamma_i^1 = \{v_y\}$ for all i, and hence, for each, the first cut is removed, which is also the cut for which ρ_i^j is not defined and for which the prerequisites for Lemma 4 do not hold. Ties between cuts coming from different T_i can be arbitrarily resolved. Write the ordered sequence $\bar{\Gamma}$ as $\{\bar{\gamma}^2, \ldots, \bar{\gamma}^r\}$. Finally, create a cut sequence $\mathcal{S}_{v_y} = \gamma^1, \ldots, \gamma^r$ for T from $\bar{\Gamma}$ as follows:

$$\gamma^l = \begin{cases} \{v_y\}, & l = 1 \\ (\gamma^{l-1} \setminus T_i) \cup \bar{\gamma}^l, & l = 2, \ldots, r \wedge \bar{\gamma}^l \subseteq T_i. \end{cases} \tag{11}$$

In colloquial terms, at each step l, we execute the cut $\bar{\gamma}^l$ in a subtree T_i, while maintaining clear all vertices from all other subtrees from the previous step $l - 1$. Note that it is necessary to remove T_i from γ^{l-1} to keep only cleared vertices in other subtrees and have cleared vertices in T_i to be exactly $\bar{\gamma}^l$ (second case in the definition earlier). This is due to the fact that we have not ruled out recontamination yet. We can now introduce the main results of this section, namely, that $\mathcal{S}_{v_y}$ is optimal.

Theorem 4: $\mathcal{S}_{v_y}$ is an optimal-cut sequence for T.

Proof: See the Appendix.

B. Constructing Cut Sets

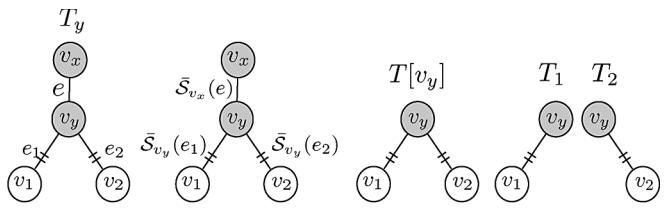
Theorem 4 provides the basis for a recursive construction of optimal-cut sequences starting at the leaves and associating them to edges, much like labels defined in Section V. Similarly to the label-based algorithm, for each starting vertex, the algorithm computes the best contiguous progressive strategy and its cost starting from that vertex. Then, looking at the costs of these n strategies, the optimal one can be retrieved. It is immediate to see that if all full-cut sequences are progressive, then so is $\mathcal{S}_{v_y}$.

A brute-force approach is not viable because we need to consider the set of all cuts for finding full-cut sequences and this set grows exponentially in the number of vertices. However, we can construct full-cut sequences more efficiently according to a bottom-up paradigm. We first show how to start the recursive construction at the leaves and then show how to find full-cut sequences efficiently.

We adopt the same perspective as in Section V, i.e., let v_x and v_y be neighbors, $e = [v_x, v_y]$, and $v_1, \ldots, v_k$ with $k = \text{degree}(v_y) - 1$ are all neighbors of v_y that are different from v_x (see Figs. 7 and 9). We now associate a full-cut sequence $\bar{\mathcal{S}}_{v_x}(e)$ to e coming from direction v_x, similar to the label $\lambda_{v_x}(e)$. Note that $\bar{\mathcal{S}}_{v_x}(e)$ is a full-cut sequence for v_x in the tree T_y that is given by removing all edges from v_x except e (see Fig. 9). If v_y is a leaf, then $k = 0$, i.e., its only neighbor is v_x. In this case, it is immediate to compute the right-cut sequence

$$\bar{\mathcal{S}}_{v_x}(e) = \{v_x\}, \{v_x, v_y\}.$$

Otherwise, if $k > 0$, we consider $v_1, \ldots, v_k$ with edges $e_i = [v_y, v_i]$, $i = 1, \ldots, k$. Let $\bar{\mathcal{S}}_{v_y}(e_i)$ be the full-cut sequence on edge e_i coming from direction v_y. By virtue of Theorem 4, we can construct an optimal-cut sequence $\mathcal{S}_{v_y}$, as defined by (11), which clears the subtree $T[v_y]$ rooted at v_y after removing e (see Fig. 9). If all v_i are leaves, this corresponds to exactly the same



$$\bar{\mathcal{S}}_{v_y}(e_1) = \{v_y\}, \{v_y, v_1\} \qquad \bar{\mathcal{S}}_{v_y}(e_2) = \{v_y\}, \{v_y, v_2\}$$

$$\mathcal{S}_{v_y} = \{v_y\}, \{v_y, v_2\}, \{v_y, v_2, v_1\}$$

Possibilities for $\bar{\mathcal{S}}_{v_x}(e)$

1) $\bar{\mathcal{S}}_{v_x}(e) = \{v_x\}, V(T_y) = \{v_x, v_y, v_2, v_1\}$

2) $\bar{\mathcal{S}}_{v_x}(e) = \{v_x\}, \{v_x, v_y\}, V(T_y)$

3) $\bar{\mathcal{S}}_{v_x}(e) = \{v_x\}, \{v_x, v_y\}, \{v_x, v_y, v_2\}, V(T_y)$

Fig. 9. Cut sequences associated with edges and the subtrees involved in the construction. There are three possibilities for $\bar{\mathcal{S}}_{v_x}(e)$, depending on the costs of the cuts, e.g., if weights on the tree are s.t. executing $\{v_x, v_y\}$ costs as much as executing the full cut $V(T_y)$ right away, then the first possibility is the full-cut sequence. Otherwise, if $\{v_x, v_y\}$ costs less and has smaller blocking cost than $\{v_x\}$, then the second possibility is the full-cut sequence, and so on.

local strategy that the algorithm from Section V produces. This is evident if one compares (4)–(10) and keeps in mind that cuts are sorted with ρ increasing.

We now need to find a full-cut sequence for T_y, where T_y is the analog of the trees T_i but from the perspective of v_x toward v_y instead of v_y toward v_i (see Fig. 9). This full-cut sequence can be associated to e and written as $\bar{\mathcal{S}}_{v_x}(e)$ to continue the recursion. The first cut is trivially $\{v_x\}$. Next, we can obtain the remaining cuts from the already available $\mathcal{S}_{v_y}$ instead of looking at all possible cuts. One observation is crucial to this construction, namely, that only cuts that correspond to a step in the execution of $\mathcal{S}_{v_y}$ need to be in $\bar{\mathcal{S}}_{v_x}(e)$. Clearly, one needs to add v_x to all cuts of $\mathcal{S}_{v_y}$, since $v_x \notin T[v_y]$. For the example with all $v_1, \ldots, v_k$ leaves, this leads to cuts $\{v_x, v_y\}, \{v_x, v_y, v_k\}, \ldots, \{v_x, v_y, v_k, \ldots, v_1\}$ (assuming that indices are ordered by ρ as in Section V). We can now obtain a full-cut sequence by selecting all cuts from this set that satisfy the criteria outlined in Definition 20. The following argument validates this claim.

Theorem 5: Let $\bar{\mathcal{S}}_{v_x}$ be a full-cut sequence for v_x in T_y and $\mathcal{S}_{v_y} = \gamma^1, \ldots, \gamma^r$ constructed as before. If $\gamma \in \bar{\mathcal{S}}_{v_x}$ and $\gamma \neq \{v_x\}$, then $\exists l \in \{1, \ldots, r\}$ s.t. $\gamma^l \in \mathcal{S}_{v_y}$ has $ag(\gamma^l \cup \{v_x\}, \nu^1) \leq ag(\gamma, \nu^1)$ and $b(\gamma^l \cup \{v_x\}) \leq b(\gamma)$.

Proof: See the Appendix.

Theorem 5 implies that we can get a full-cut sequence $\bar{\mathcal{S}}_{v_x}$ to associate to e by only considering the cuts arising from $\mathcal{S}_{v_y}$. Algorithm 1 shows how to use the results from Theorems 4 and 5 to compute an optimal strategy. Just like we did with labels before, the algorithm recursively builds cut sequences on the edges of a surveillance tree T with two directions for each edge. To finally obtain $ag(T)$, we construct $\mathcal{S}_{v_y}$ on T for each vertex $v_y \in V(T)$, and then the vertex with the lowest cost is selected as the starting vertex. This is similar to the procedure in Section V. Algorithm 1 presents this in pseudocode and returns $ag(T)$. Once the best vertex v_y is found, translating the cut sequence $\mathcal{S}_{v_y}$ into a strategy is straightforward. In fact, by following the

1: Set all $\bar{\mathcal{S}}_v(e)$ to $\emptyset$ and initialize empty queue Q
2: $Q.enqueue(leaves(T))$
3: **while not** $Q.empty()$ **do**
4: $\quad v_y \leftarrow Q.dequeue()$
5: $\quad$ **if** $degree(v_y) = 1$ **then**
6: $\quad\quad v_x \leftarrow neighbors(v_y)$
7: $\quad\quad \bar{\mathcal{S}}_{v_x}([v_x, v_y]) \leftarrow \{\{v_x\}, \{v_x, v_y\}\}$
8: $\quad$ **else if** $\exists$ neighbor v_x s.t. $\bar{\mathcal{S}}_{v_x}([v_x, v_y]) = \emptyset$ **then**
9: $\quad\quad k \leftarrow degree(v_y) - 1$
10: $\quad\quad$ Let $v_1, \ldots, v_k$ be neighbors s.t. $\bar{\mathcal{S}}_{v_y}([v_y, v_i]) \neq \emptyset$
11: $\quad\quad$ Construct $\mathcal{S}_{v_y}$ on $T[v_y]$
12: $\quad\quad$ Construct $\bar{\mathcal{S}}_{v_x}$ on T_y from $\mathcal{S}_{v_y}$
13: $\quad\quad \bar{\mathcal{S}}_{v_x}([v_x, v_y]) \leftarrow \bar{\mathcal{S}}_{v_x}$
14: $\quad a \leftarrow$ number of neighbors of v_x s.t. $\bar{\mathcal{S}}_{v_x}([v_x, v]) \neq \emptyset$
15: $\quad$ **if** $a = degree(v_x) - 1$ **then**
16: $\quad\quad Q.enqueue(v_x)$
17: $\quad$ **else if** $a = degree(v_x)$ **then**
18: $\quad\quad$ **for all** $v \in neighbors(v_x)$ s.t. $\bar{\mathcal{S}}_v([v, v_x]) = \emptyset$ **do**
19: $\quad\quad\quad Q.enqueue(v_x)$
20: **for all** $v \in V(T)$ **do**
21: $\quad$ Construct $\mathcal{S}_v$ on T
22: $\quad ag(v) \leftarrow c(\mathcal{S}_v)$
23: **return** $\min_{v \in V(T)}(ag(v))$

Algorithm 1. Cut_strategy (T).

edges and using the associated cut sequences, one can write $\mathcal{S}_{v_y}$ as a simple-cut sequence that can be immediately converted into a strategy.

The complexity of algorithm 1 is $O(n^2)$ and can be computed as follows. Throughout the analysis, it is important to note that since we are dealing with a tree, the number of edges is $m = n - 1$; therefore, eventually, we compute the complexity as a function of the number of vertices only. Clearly, line 11 is the most costly part in which we have to sort cuts when constructing $\mathcal{S}_{v_y}$ on $T[v_y]$. However, the number of cuts to be sorted is bounded by n due to the linear construction, thus leading to a contribution of at most one cut for a vertex in the tree. This is obvious from the fact that $|\bar{\mathcal{S}}_{v_x}| \leq 1 + |\mathcal{S}_{v_y}| = 2 + \sum_{i=1}^{k} |\bar{\mathcal{S}}_i| - 1$, but even better, each $\bar{\mathcal{S}}_i$ is already sorted by ρ, therefore, to construct $\mathcal{S}_{v_y}$, we have to merge $\mathrm{degree}(v_y) - 1$ sequences that altogether are at most of length n, which can be done in $O(\log(\mathrm{degree}(v_y)) \times n)$. Line 11 is executed twice for each edge once in each direction, or in other words, $\mathrm{degree}(v_y)$ times for each vertex v_y. However, the only difference between two executions of line 11 at vertex v_y is that the full-cut sequence from one edge e_i is replaced by the full-cut sequence on another edge. More precisely, the edge e from v_x to v_y of a previous execution of line 11 becomes one of the edges e_i toward v_i in a subsequent execution while one of the previous e_i becomes e. Hence, we can reuse the sorted $\mathcal{S}_{v_y}$ of the first execution of line 11 for subsequent executions by just removing one of the cut sequences in a T_i and adding one. This can be done linearly in n. Therefore, we only need to merge $\mathrm{degree}(v_y)$ sorted sequences once for each vertex, which leads to $\sum_{v_y \in V(T)} \log(\mathrm{degree}(v_y)) \times n \leq 2m \times n$, and then

reuse it for the $\mathrm{degree}(v_y) - 1$ remaining executions, which can be done in $\sum_{v_y \in V(T)} (\mathrm{degree}(v_y) - 1) \times n \leq 2m \times n$. Hence, the overall complexity is $O(n^2)$.

VIII. Conclusion

In this paper, we presented a novel theoretical framework to model surveillance tasks performed by multiple robots with limited sensing capabilities. The approach we presented has two major advantages. First, it produces coordination plans, called strategies, that abstract from low-level sensing details. Limited sensing capabilities of robots in the team are accounted for by assuming that multiple robots are needed in order to perform the basic operations, i.e., blocking a connection between two areas, or sweeping an area. Second, by formalizing it into a well-characterized graph-optimization problem, we were able to leverage a significant amount of former graph-related literature and gain significant insights into its computational structure. After having established the formal framework and determined its computational complexity, we turned our attention to the tractable case of trees. Surveillance graphs can be easily turned into surveillance trees by permanently blocking edges that lead to cycles in the graph. This approach is simple but not very effective, and in [34], we have shown how to attack this problem more effectively. We presented an algorithm for the special case of trees and contiguous strategies that is optimal. Contiguous strategies are more restricted, and therefore, in general, one can expect strategies that are not required to be contiguous to need fewer robots. The existence of a polynomial algorithm capable of producing an optimal noncontiguous strategy for trees is an open question. In fact, in [37], we have recently discovered a hybrid method that combines previous contiguous and noncontiguous labeling methods and produces noncontiguous strategies that in certain situations can outperform contiguous strategies. This hybrid method, however, is based on a dynamic programming approach and its complexity is pseudopolynomial. This tradeoff therefore justifies the use of the optimal algorithm for contiguous strategies presented in this paper with polynomial complexity. Contiguous strategies are not only interesting from a mere theoretical point of view. For example, in certain situations, recontamination of a cleared room should be avoided because it could be used to deploy infrastructures that would be negatively impacted by intruders.

The framework presented in this paper provides a solid foundation for future research in this area but also raises some questions unanswered at the moment. The following are some of the issues we believe to be most important. First, throughout the paper, we have assumed a deterministic scenario, i.e., we did not contemplate the possibility of faulty sensors, such as sensors affected by false positive or negatives. This problem is currently being investigated, and first partial answers have recently been found [38]. Second, we have defined Graph-Clear with the objective of minimizing the number of robots needed to detect all intruders, but one could aim for the optimization of different parameters. For example, if one considers the motion model of the robots being used, then one could look for strategies that are fast to execute, minimize energy consumption, etc. Finally, one issue is the detailed implementation of sweep operations for a given class of robots and sensors. This problem is currently being investigated, and preliminary results have been submitted for publication.

Appendix

Proof of Lemma 3: Let $\mathcal{S}_c = \{a_1, a_2, \ldots, a_r\}$, and let $\gamma^1, \ldots, \gamma^r$ be the subsets of vertices cleared by $\mathcal{S}_c$ during the execution of its r steps. By Lemma 1, we can assume that at most one vertex is cleared at each step of $\mathcal{S}_c$, and by hypothesis, $\mathcal{S}_c$ is contiguous. Therefore, $\mathcal{S} = \gamma^1, \ldots, \gamma^r$ is a simple-cut sequence in T. Compare (2) with (9) and (1) with (8). By substitution, one can verify that $c(a_i) = f(\gamma^i)$ $(1 \leq i \leq r)$, and then, $f(\mathcal{S}) \leq k$. □

The following lemma is needed in order to prove Theorem 2. Its simple proof can be found in [3].

Lemma 5: Let γ^1 and γ^2 be two cuts of a surveillance tree $T = (V, E, w)$. Then

$$b(\gamma^1 \cup \gamma^2) + b(\gamma^1 \cap \gamma^2) \leq b(\gamma^1) + b(\gamma^2).$$

Proof of Theorem 2: Out of all simple-cut sequences with frontier not greater than k, choose $\mathcal{S} = \gamma^1, \ldots, \gamma^r$ satisfying the following properties.

1) $\sum_j f(\gamma^j)$ is minimal.
2) $\sum_j |\gamma^j|$ is minimal, subject to the previous constraint.

We will now show that such $\mathcal{S}$ is a progressive simple-cut sequence. This means that

a) $|\gamma^i \setminus \gamma^{i-1}| = 1$ for all $2 \leq i \leq r$;
b) $\gamma^1 \subseteq \gamma^2 \subseteq \cdots \subseteq \gamma^r$.

It is immediate seen that the property a) holds for $\mathcal{S}$. In fact, it cannot be the case that $|\gamma^i \setminus \gamma^{i-1}| = 0$, because, otherwise, the simple-cut sequence obtained from $\mathcal{S}$ by excluding γ^i would invalidate property 1. Therefore, $|\gamma^i \setminus \gamma^{i-1}| = 1$.

We now show that property b) holds for $\mathcal{S}$ as well. First, for an arbitrary index i, let us consider $\gamma^* = \gamma^{i-1} \cup \gamma^i$. If $f(\gamma^*) < f(\gamma^i)$, then $\mathcal{S}^* = \gamma^1, \ldots, \gamma^{i-1}, \gamma^*, \gamma^{i+1}, \ldots, \gamma^r$ would be a simple-cut sequence violating property 1. Therefore

$$f(\gamma^*) \geq f(\gamma^i).$$

With some work, we can now derive a similar relationship involving $b(\gamma^*)$ and $b(\gamma^i)$. Since property a) holds, $v_i = \gamma^i \setminus \gamma^{i-1}$ is always well defined, i.e., $v_i \neq \emptyset$. Let $v^* = \gamma^* \setminus \gamma^{i-1}$, and rewrite the previous inequality using (8) explicitly

$$b(\gamma^*) + s(v^*) - \sum_{e \in \delta v^* \cap \delta \gamma^*} w(e)$$
$$\geq b(\gamma) + s(v_i) - \sum_{e \in \delta v_i \cap \delta \gamma^i} w(e).$$

By construction $v^* = v_i$, and therefore, the inequality simplifies to

$$b(\gamma^*) - \sum_{e \in \delta v^* \cap \delta \gamma^*} w(e) \geq b(\gamma^i) - \sum_{e \in \delta v_i \cap \delta \gamma^i} w(e). \qquad (12)$$

In order to further simplify the expression, let us observe that there is exactly one edge between γ^{i-1} and v_i. If this was not the case, there would be a cycle in the tree T. Let e' be this unique

edge. By construction $\gamma^i \subseteq (\{v_i\} \cup \gamma^{i-1})$, and therefore, e' is the only edge in $\text{edges}(v_i)$ with both extremes in γ^i. The exact same reasoning applies to γ^*, and hence, we get (remember $v^* = v_i$)

$$\sum_{e \in \delta v^* \cap \delta \gamma^*} w(e) = \sum_{e \in \delta v_i \cap \delta \gamma^i} w(e).$$

Inequality (12) then reduces to $b(\gamma^*) \geq b(\gamma^i)$, i.e., what we wanted. Let us turn our attention to $v_{i-1} = \gamma^{i-1} \setminus \gamma^{i-2}$. If $v_{i-1} \notin \gamma^i$, then $\gamma^1, \ldots, \gamma^{i-2}, \gamma^i, \ldots, \gamma^r$ is a simple-cut sequence, violating property 1. Therefore $v_{i-1} \in \gamma^i$. Now consider the set $\gamma^{**} = \gamma^{i-1} \cap \gamma^i$. Since v_{i-1} belongs to both γ^{i-1} and γ^i, then $\gamma^{**} \neq \emptyset$, and γ^{**} is connected. Then, using inequality (12) while applying Lemma 5, we can then conclude that $b(\gamma^{**}) \leq b(\gamma^{i-1})$. Now, consider the cut sequence

$$\mathcal{S}^{**} = \gamma^1, \ldots, \gamma^{i-2}, \gamma^{**}, \gamma^i, \ldots, \gamma^r.$$

$\mathcal{S}^{**}$ is a simple-cut sequence. Moreover, it is easy to show that $f(\gamma^{**}) \leq f(\gamma^{i-1})$. Start with

$$\begin{aligned} f(\gamma^{**}) &= b(\gamma^{**}) + s(v_{i-1}) - \sum_{e \in \delta v_{i-1} \cap \delta \gamma^{**}} w(e) \\ &\leq b(\gamma^{i-1}) + s(v_{i-1}) - \sum_{e \in \delta v_{i-1} \cap \delta \gamma^{**}} w(e). \end{aligned}$$

By simple set relations, it follows that $\delta v_{i-1} \cap \delta \gamma^{i-1} \subseteq \delta v_{i-1} \cap \delta \gamma^{**}$, and then, we get

$$f(\gamma^{**}) \leq b(\gamma^{i-1}) + s(v_{i-1}) - \sum_{e \in \delta v_{i-1} \cap \delta \gamma^{i-1}} w(e).$$

The right side of this inequality is $f(\gamma^{i-1})$, then

$$f(\gamma^{**}) \leq f(\gamma^{i-1}) \leq k.$$

Therefore, $\mathcal{S}^{**}$ is a simple-cut sequence with frontier smaller or equal than k. Moreover, it must be that $|\gamma^{i-1} \cap \gamma^i| \geq |\gamma^{i-1}|$; otherwise, we would violate property 2 with $\mathcal{S}^{**}$, but $|\gamma^{i-1} \cap \gamma^i| \geq |\gamma^{i-1}|$ implies that $\gamma^{i-1} \subseteq \gamma^i$, and then, we have proven property b) as well, thus completing the proof. □

Proof of Theorem 3: By the previous lemma and theorem, the existence of a progressive simple-cut sequence with frontier not greater than $ag(\mathcal{S}_c)$ is guaranteed. Let $\mathcal{S} = \gamma^1, \ldots, \gamma^r$ be this progressive-cut sequence. For $2 \leq i \leq r$, let $v_i = \gamma^i \setminus \gamma^{i-1}$, and let v_1 be the only element in γ^1. $\mathcal{S}$ leads directly to a contiguous progressive strategy by clearing the vertices v_i in order. First, consider v_1. By simple substitution, $f(\gamma^1) = s(v_1)$. Assume that γ^i is cleared with cost $f(\gamma^i)$. At the end of the step $b(\gamma^i)$, agents are required to avoid recontamination of γ^i. Adding v_{i+1} to γ^i has cost $s(v_{i+1}) - \sum_{e \in \delta v_{i+1} \cap \delta \gamma^i} w(e)$, which leads to $b(\gamma^i) + s(v_{i+1}) - \sum_{e \in \delta v_{i+1} \cap \delta \gamma^i} w(e) = b(\gamma^{i+1}) - \sum_{e \in \delta v_{i+1} \cap \delta \gamma^{i+1}} w(e) = f(\gamma^{i+1})$ agents. Therefore, a progressive contiguous clearing strategy of cost not greater than $ag(\mathcal{S}_c)$ exists. Since we started assuming $\mathcal{S}_c$ is optimal, then so is $ag(\mathcal{S}_c)$, which concludes the proof. □

Proof of Lemma 4: The claim is true for $l = b$ by substitution. Let us now assume that $ag(\gamma^l, \nu^l) = ag(\gamma^l, \nu^b)$ for a certain value of l in the range $b \leq l < r$ and prove that $ag(\gamma^{l+1}, \nu^{l+1}) = ag(\gamma^{l+1}, \nu^b)$. By definition $ag(\gamma^{l+1}, \nu^b)$ is the cost of the optimal strategy that removes all contamination from γ^{l+1} in state ν^b. Consider the following strategy. Start with T in state ν^b and execute γ^l. This will change the state of the tree to ν^{l+1}. Then, execute γ^{l+1} starting from the current state ν^{l+1}. The cost of this strategy is

$$ag(\gamma^{l+1}, \nu^b) = \max((ag(\gamma^l, \nu^b), ag(\gamma^{l+1}, \nu^{l+1})).$$

This maximum cannot be $ag(\gamma^l, \nu^b)$ since by assumption $ag(\gamma^l, \nu^b) = ag(\gamma^l, \nu^l) < ag(\gamma^{l+1}, \nu^{l+1})$. Therefore, $ag(\gamma^{l+1}, \nu^b) = ag(\gamma^{l+1}, \nu^{l+1})$. □

Proof of Theorem 4: We started assuming that v_y is the first vertex cleared by an optimal progressive contiguous strategy for T, so starting with $\gamma^1 = \{v_y\}$ does not compromise the possibility of getting an optimal strategy. Let us consider $l \geq 2$. By construction, every cut γ^l has an associated cut $\bar{\gamma}^l$ [see (11)]. Such $\bar{\gamma}^l$ was constructed from a certain $\gamma_i^j \in \bar{\mathcal{S}}_i$. We can therefore associate each cut γ^l with $l \geq 2$ with a couple of indexes i and j such that γ^l originated from γ_i^j.

Let us now describe the costs of $\mathcal{S}_{v_y}$ and relate it to the costs in the full-cut sequences for each T_i. $b(\gamma^l)$ is the blocking cost in T after executing γ^l. The part of this blocking cost in T_i is given by $b_i^l = b(\gamma^l \cap T_i)$ and is equal to $b(\gamma_i^j)$ by construction. The cost of execution of γ^l is given by $c^l = ag(\gamma^l, \nu^l)$. We can relate c^l to the costs inside each subtree T_i with the following relationship:

$$c^l = b^{l-1} - b_i^{l-1} + ag(\gamma_i^j, \nu_i^j). \tag{13}$$

Notice that ν_i^j of the cut sequence $\bar{\mathcal{S}}_i$ is identical to the state that ν^l of the cut sequence $\mathcal{S}_{v_y}$ induces on T_i. Equation (13) follows simply by construction. It results from keeping $(\gamma^{l-1} \setminus T_i)$ blocked, which costs $b^{l-1} - b_i^{l-1}$, and executing γ_i^j in T_i with cost $ag(\gamma_i^j, \nu_i^j)$. By Lemma 4 and the observation that $b_i^{l-1} = b(\gamma_i^{j-1})$, i.e., the blocking cost from the cut in $\bar{\mathcal{S}}_i$ right before γ_i^j, we get [see (10)]

$$c^l = b^{l-1} - b_i^{l-1} + ag(\gamma_i^j, \nu_i^2) = b^{l-1} + \rho_i^j. \tag{14}$$

Here, the significance of ρ values formerly defined becomes apparent. Notice the similarity to ordering subtrees in Section V. In colloquial terms, the ordering with ρ increasing asks for the next cut that can reduce the blocking cost while not costing much to execute.

Now, let $\hat{\mathcal{S}}$ be an optimal contiguous strategy that is progressive and starts at v_y. We will adopt a similar notation as for $\mathcal{S}_{v_y}$ but adding $\hat{}$ to all terms. Due to Lemma 1, we can assume that $\hat{\mathcal{S}}$ clears exactly one new vertex per step. Therefore, it can be written as a simple progressive cut sequence $\hat{\gamma}^1, \ldots, \hat{\gamma}^n$ with exactly n cuts. It follows that for every $l = 2, \ldots, n$, we have one and only one i s.t. $\hat{\gamma}^l \setminus \hat{\gamma}^{l-1} \subset T_i$. This allows us to consider, for each T_i, a cut sequence given by all nonempty $\hat{\gamma}^l \cap T_i$ for all l s.t. $\hat{\gamma}^l \setminus \hat{\gamma}^{l-1} \subset T_i$. To identify these cut sequences restricted to each T_i, we use $\hat{\gamma}_i^j = \hat{\gamma}^l \cap T_i$, again associating each step l in $\hat{\mathcal{S}}$ with a $\hat{\gamma}_i^j$. Hence, a similar analysis as earlier for $\mathcal{S}_{v_y}$ applies, and we get

$$\hat{c}^l = \hat{b}^{l-1} - \hat{b}_i^{l-1} + ag(\hat{\gamma}_i^j, \hat{\nu}_i^l) \tag{15}$$

where now $\hat{\nu}_i^l$ is simply the state of T_i induced by $\hat{\nu}^l$ and equal to $\hat{\nu}_i^j$, which is the state of T_i after execution of $\hat{\gamma}_i^1, \ldots, \hat{\gamma}_i^{j-1}$ in T_i. Now that we can describe $\mathcal{S}_{v_y}$ and $\hat{\mathcal{S}}$, let us compare the two and their costs. We will do so by replacing cuts in the cut sequences $\hat{\gamma}_i^j$ with cuts from the full-cut sequences used to construct $\mathcal{S}_{v_y}$. In colloquial terms, we will show that cuts from the full-cut sequences are not more costly than the optimal ones. For each i, consider the cut sequence $\hat{\gamma}_i^j$ in T_i with associated clearing cost $ag(\hat{\gamma}_i^j, \hat{\nu}_i^j)$ and blocking cost $b(\hat{\gamma}_i^j)$. First, we want to remove all cuts $\hat{\gamma}_i^j$ that do not reduce b^l, i.e., have $b(\hat{\gamma}_i^j) \geq b(\hat{\gamma}_i^{j-1})$. It is evident from (15) that this removal does not increase b^l at any step. Hence, removal of such a cut $\hat{\gamma}_i^j$ can only lead to larger costs if it increases the cost for a subsequent cut, i.e., $ag(\hat{\gamma}_i^p, \hat{\nu}_i^p)$, for some $p > j$, but, if after removal of $\hat{\gamma}_i^j$ we have $ag(\hat{\gamma}_i^p, \hat{\nu}_i^p)$ larger than before, then (through a similar argument as for the proof of Lemma 4) we have $ag(\hat{\gamma}_i^p, \hat{\nu}_i^p) \leq ag(\hat{\gamma}_i^j, \hat{\nu}_i^j)$ and, hence, no overall larger cost. Therefore, we can remove all such cuts without increasing the overall cost, which leads to b^l becoming a strictly decreasing sequence.

With a similar argument, we can modify the sequence $\hat{\gamma}_i^j$ to have $ag(\hat{\gamma}_i^j, \hat{\nu}_i^j)$ strictly increasing. Notice that if $ag(\hat{\gamma}_i^j, \hat{\nu}_i^j) \geq ag(\hat{\gamma}_i^{j+1}, \hat{\nu}_i^{j+1})$, then removal of $\hat{\gamma}_i^j$ and executing $\hat{\gamma}_i^{j+1}$ instead will not increase costs c^l, since $b(\hat{\gamma}_i^j) > b(\hat{\gamma}_i^{j+1})$, and $ag(\hat{\gamma}_i^{j+1}, \hat{\nu}_i^j) \leq ag(\hat{\gamma}_i^j, \hat{\nu}_i^j)$.

After these removals, we are in a condition that satisfies the hypothesis of Lemma 4 and obtain

$$\hat{c}^l = \hat{b}^{l-1} - \hat{b}_i^{l-1} + ag(\hat{\gamma}_i^j, \hat{\nu}_i^2) = \hat{b}^{l-1} - \hat{\rho}_i^j. \tag{16}$$

It is now clear to see (looking at ρ) that replacing every cut $\hat{\gamma}_i^j$ with a cut from the full-cut sequence with equal or smaller ag and adding all remaining full cuts, ordered by ρ, leads to no increased cost. Hence, we can turn $\hat{\mathcal{S}}$ into $\mathcal{S}_{v_y}$ without incurring larger cost, and hence, $\mathcal{S}_{v_y}$ is optimal. □

Proof of Theorem 5: Let $\gamma \in \bar{\mathcal{S}}_{v_x}$. By definition $v_x \in \gamma$. If $\gamma = \{v_x, v_y\}$, the claim is trivially true by noting that $l = 1$ with $\gamma^1 = \{v_y\}$, and then, $ag(\{v_y\} \cup \{v_x\}, \nu^1) = ag(\gamma, \nu^1)$.

Otherwise, γ has one or more vertices in some T_i, $i = 1, \ldots, k$. We can hence write the execution of γ as a new sequence of cuts starting at $\hat{\gamma}^1 = \{v_x\}, \hat{\gamma}^2 = \{v_x, v_y\}$ and continuing with cuts separated into T_i similar as for the proof of Theorem 4 . Write $\hat{\gamma}^3, \ldots, \hat{\gamma}^t = \gamma$ for these cuts. Note that this is not a cut sequence for T_y but a only a sequence of cuts because the full cut is missing. Again, as for Theorem 4, $\hat{\gamma}^3, \ldots, \hat{\gamma}^t$ induces sequences of cuts in the subtrees T_i, written $\hat{\gamma}_i^j$. The only difference to the proof for Theorem 4 is that we had two steps $\hat{\gamma}^1$ and $\hat{\gamma}^2$ prior to considering the cuts in subtrees T_i, which means that v_x and v_y are both not going to be recontaminated. Therefore, we can also apply the same replacement method as for Theorem 4 and all cuts $\hat{\gamma}_i^j$ can be replaced with cuts from full cut sequences of T_i without incurring larger costs.

Now, all $\hat{\gamma}_i^j$ are cuts that also appear in the full-cut sequences for T_i. The only significant difference to the proof of Theorem 4 is that after this replacement, there is a last cut in each sequence $\hat{\gamma}_i^j$, which is not necessarily a full cut for T_i, and therefore, the sequence is still not a cut sequence. Now, the last cut $\hat{\gamma}^t$ is associated with a last cut of some sequence $\hat{\gamma}_i^j$ in some T_i, which is written $\hat{\gamma}_i^p$. Since $\hat{\gamma}_i^p \in \bar{\mathcal{S}}_i$ due to the replacement method, we have a cut γ^l in $\mathcal{S}_{v_y}$ associated with it as well. By construction and from (14), $\mathcal{S}_{v_y}$ incurs no higher cost up until γ^l than the sequence $\hat{\gamma}^3, \ldots, \hat{\gamma}^t$. This is due to the fact that $\mathcal{S}_{v_y}$ is based on all cuts from the full-cut sequences $\bar{\mathcal{S}}_i$ for each T_i while $\hat{\gamma}^3, \ldots, \hat{\gamma}^t$ may have some cuts omitted. From (14), it is clear that adding additional cuts from the full-cut sequences $\bar{\mathcal{S}}_i$ can only improve the costs since those cuts with ρ lowest are executed first and after being executed can only decrease the overall blocking cost. It follows that $ag(\gamma^l \cup \{v_x\}, \nu^1) \leq ag(\gamma, \nu^1)$ and that $b(\gamma^l \cup v_x) = b(\gamma^l) \leq b(\gamma)$. □

ACKNOWLEDGMENTS

This paper extends and complements preliminary findings published in [34], [36], and [37]. The authors thank the reviewers for constructive feedback.

REFERENCES

[1] A. Kolling and S. Carpin, "Extracting surveillance graphs from robot maps," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2008, pp. 2323–2328.

[2] T. Parsons, "Pursuit-evasion in a graph," in *Theory and Application of Graphs*, vol. 642, Y. Alavi and D. R. Lick, Eds. Berlin/Heidelberg, Germany: Springer-Verlag, 1976, pp. 426–441.

[3] L. Barriere, P. Flocchini, P. Fraigniaud, and N. Santoro, "Capture of an intruder by mobile agents," in *Proc. 14th Annu. ACM Symp. Parallel Algorithms Archit*. New York: ACM, 2002, pp. 200–209.

[4] S. Sachs, S. Rajko, and S. LaValle, "Visibility-based pursuit-evasion in an unknown planar environment," *Int. J. Robot. Res.*, vol. 23, no. 1, pp. 3–26, Jan. 2004.

[5] A. Kolling and S. Carpin, "Surveillance strategies for target detection with sweep lines," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2009, pp. 5821–5827.

[6] I. Suzuki and M. Yamashita, "Searching for a mobile intruder in a polygonal region," *SIAM J. Comput.*, vol. 21, no. 5, pp. 863–888, 1992.

[7] M. Yamashita, H. Umemoto, I. Suzuki, and T. Kameda, "Searching for mobile intruders in a polygonal region by a group of mobile searchers," in *Proc. Symp. Comput. Geom.*, 1997, pp. 448–450.

[8] S. LaValle, D. Lin, L. Guibas, J. Latombe, and R. Motwani, "Finding an unpredictable target in a workspace with obstacles," in *Proc. IEEE Int. Conf. Robot. Autom.*, 1997, pp. 737–742.

[9] L. J. Guibas, J.-C. Latombe, S. M. LaValle, D. Lin, and R. Motwani, "A visibility-based pursuit-evasion problem," *Int. J. Comput. Geom. Appl.*, vol. 9, pp. 471–494, 1999.

[10] S.-M. Park, J.-H. Lee, and K.-Y. Chwa, "Visibility-based pursuit-evasion in a polygonal region by a searcher," *Lect. Notes Comput. Sci.*, vol. 2076, pp. 456–468, 2001.

[11] S. M. LaValle and J. Hinrichsen, "Visibility-based pursuit-evasion: The case of curved environments," *IEEE Trans. Robot. Autom.*, vol. 17, no. 2, pp. 196–201, Apr. 2001.

[12] S. LaValle, B. Simov, and G. Slutzki, "An algorithm for searching a polygonal region with a flashlight," *Int. J. Comput. Geom.*, vol. 12, no. 1/2, pp. 87–113, 2002.

[13] L. Guilamo, B. Tovar, and S. LaValle, "Pursuit-evasion in an unknown environment using gap navigation trees," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Apr. 2004, vol. 4, pp. 3456–3462.

[14] B. Tovar, L. Guilamo, and S. M. LaValle, "Gap navigation trees: Minimal representation for visibility-based tasks," in *Proc. Workshop Algorithmic Found. Robot.*, 2004, vol. 17, pp. 3855–3862.

[15] B. Tovar, S. M. LaValle, and R. Murrieta., "Locally-optimal navigation in multiply-connected environments without geometric maps," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2003, vol. 3, pp. 3491–3497.

[16] B. Simov, G. Slutzki, and S. LaValle, "Clearing a polygon with two 1-searchers," *Int. J. Comput. Geom. Appl.*, vol. 19, no. 1, pp. 59–92, 2009.

[17] B. Tovar and S. M. LaValle, "Visibility-based pursuit-evasion with bounded speed," presented at the Workshop Algorithmic Found. Robot., New York, 2006.

[18] J. Yu and S. LaValle, "Tracking hidden agents through shadow information spaces," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2008, pp. 2331–2338.
[19] F. V. Fomin and D. M. Thilikos, "An annotated bibliography on guaranteed graph searching," *Theor. Comput. Sci.*, vol. 399, no. 3, pp. 236–245, 2008.
[20] N. Megiddo, S. L. Hakimi, M. R. Garey, D. S. Johnson, and C. H. Papadimitriou, "The complexity of searching a graph," *J. ACM*, vol. 35, no. 1, pp. 18–44, 1988.
[21] M. R. Garey and D. S. Johnson, *Computers and Intractability : A Guide to the Theory of NP-Completeness*. New York: Freeman, 1979.
[22] A. S. LaPaugh, "Recontamination does not help to search a graph," *J. ACM*, vol. 40, no. 2, pp. 224–245, 1993.
[23] D. Bienstock and P. Seymour, "Monotonicity in graph searching," *J. Algorithms*, vol. 12, no. 2, pp. 239–245, 1991.
[24] M. Kirousis and C. H. Papadimitriou, "Searching and pebbling," *Theor. Comput. Sci.*, vol. 47, no. 2, pp. 205–218, 1986.
[25] F. V. Fomin, D. Kratsch, and H. Müller, "On the domination search number," *Discrete Appl. Math.*, vol. 127, no. 3, pp. 565–580, 2003.
[26] F. Makedon and I. H. Sudborough, "On minimizing width in linear layouts," *Discrete Appl. Math.*, vol. 23, no. 3, pp. 243–265, 1989.
[27] J. A. Ellis, I. H. Sudborough, and J. S. Turner, "The vertex separation and search number of a graph," *Inf. Comput.*, vol. 113, no. 1, pp. 50–79, 1994.
[28] K. Skodinis, "Computing optimal linear layouts of trees in linear time," in *ESA 2000: Proc. 8th Annu. Eur. Symp. Algorithms*. London, U.K.: Springer-Verlag, 2000, pp. 403–414.
[29] L. Parker, "Distributed algorithms for multi-robot observation of multiple moving targets," *Auton. Robots*, vol. 12, pp. 231–255, 2002.
[30] A. Kolling, S. Carpin, "Cooperative observation of multiple moving targets: An algorithm and its formalization," *Int. J. Robot. Res.*, vol. 29, no. 9, pp. 935–953, 2007.
[31] B. Jung and G. S. Sukhatme, "Tracking targets using multiple mobile robots: The effect of environment occlusion," *Auton. Robots*, vol. 13, no. 2, pp. 191–205, 2002.
[32] M. Moors, T. Röhling, and D. Schulz, "A probabilistic approach to coordinated multi-robot indoor surveillance," in *Proc IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2005, pp. 3447–3452.
[33] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Boston, MA: McGraw-Hill, 2001.
[34] A. Kolling and S. Carpin, "The graph-clear problem: Definition, theoretical properties and its connections to multirobot aided surveillance," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2007, pp. 1003–1008.
[35] V. V. Vazirani, *Approximation Algorithms*. New York: Springer-Verlag, 2001.
[36] A. Kolling and S. Carpin, "Detecting intruders in complex environments with limited range mobile sensors," in *Robot Motion and Control*, K. Kozlowski, Ed., Lect. Notes Control Inf. Sci., vol. 360. New York: Springer-Verlag London, 2007, pp. 41–426.
[37] A. Kolling and S. Carpin, "Multi-robot surveillance: An improved algorithm for the graph-clear problem," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2008, pp. 2360–2365.
[38] A. Kolling and S. Carpin, "Probabilistic graph clear," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2009, pp. 3508–3514.

Andreas Kolling (S'05) received the B.S. degree in mathematics and the M.S. degree in computer science from Jacobs University, Bremen, Germany, in 2004 and 2006, respectively. He is currently working toward the Ph.D. degree in electrical engineering and computer science at the University of California, Merced.

His research interests include multirobot systems, pursuit-evasion, cooperative robotics, and machine learning.

Stefano Carpin (S'02–M'03) received the Laurea (M.Sc.) and Ph.D. degrees in electrical engineering and computer science from the University of Padua, Padua, Italy, in 1999 and 2003, respectively.

From 2003 to 2006, he held faculty positions with Jacobs University, Bremen, Germany. Since 2007, he has been an Assistant Professor with the School of Engineering, Univresity of California, Merced, where he established and leads the Robotics Laboratory. From 2006 to 2009, he served as an Elected Executive Member of the RoboCup Federation. Under his supervision, teams participating in the RoboCupRescue Virtual Robots Competition won Second Place in 2006 and 2008 and First Place in 2009. His research interests include mobile and cooperative robotics for service tasks and robot algorithms.