Multi-robot Surveillance: an Improved Algorithm for the GRAPH-CLEAR Problem

Andreas Kolling Stefano Carpin
School of Engineering
University of California, Merced
Merced – USA

***Abstract*— The main contribution of this paper is an improved algorithm for the GRAPH-CLEAR problem, a novel NP-complete graph theoretic problem we recently introduced as a tool to model multi-robot surveillance tasks. The proposed algorithm combines two previously developed solving techniques and produces strategies that require less robots to be executed. We provide a theoretical framework useful to identify the conditions for the existence of an optimal solution under special circumstances, and a set of mathematical tools characterizing the problem being studied. Finally we also identify a set of open questions deserving more investigations.**

I. Introduction

The use of multi-robot systems for the surveillance of vast regions is one of the well established areas in multi-robot research. Up to now, however, there have been still very few on-field deployments of these systems for real world applications. Besides the obvious matter of cost, another reason for their moderate use is the fact that many basic questions about the efficient coordination of these systems are still unanswered. A big fraction of former theoretical research developed models where robots were equipped with sensors abstractions pretty far from realistic applications, e.g. sensors with infinite range and the alike. In this paper we instead extend our previous findings aimed to investigate surveillance tasks by multi-robot systems where individual agents use sensors with limited capabilities. We started this research thread with two papers [5][6] aimed to extend the CMOMMT (Cooperative Multi-robot Observation of Multiple Moving Targets) problem initially posed by Parker [11]. One of the main limitations of these algorithms is the requirement that robots operate in open areas. Our following efforts have therefore been devoted to scenarios where robots operate in cluttered environments [7]. In particular, we modeled the problem of discovering multiple intruders in a complex environment using a novel graph theoretic problem, dubbed GRAPH-CLEAR. Informally speaking, the problem asks what is the minimum number of robots needed to detect all possible intruders in a given complex environment that can be modeled as a graph. In [8] we proved that the associated decision problem is NP-complete. As clarified later on, a way to circumvent the intractability of the problem on graphs, is to perform certain *guard* operations that turn graphs into trees. In [7] and [8] we have provided two algorithms that produce search strategies for trees, i.e. course of actions for a robot team that ensures each intruder will be discovered. Both algorithms are known to be suboptimal. In this paper we present a new approach for the GRAPH-CLEAR problem restricted to trees that outperforms the previous ones. It is worth to outline that many of the properties regarding the GRAPH-CLEAR problem restricted to trees are still to be investigated. For example, we do not know yet whether such restriction to trees allows to find the optimal solution in polynomial time. This paper, however, provides a further improvement that sheds some more light on this problem, and provides some more formalism that could be used to answer this question and similar ones.

The paper is organized as follows. In section II we revise former research related to multi-robot surveillance, and we provide references to seminal papers on graph theory related to the problem at hand. Section III summarizes the GRAPH-CLEAR problem and shortly addresses our formerly developed algorithms. The new approach and a theoretical framework are presented in section IV, followed by a new algorithm that computes strategies for the new approach under certain conditions. Section V concludes with a discussion of remaining problems and possible extensions of the presented work.

II. Related Research

Visibility-based pursuit evasion games have attracted remarkable attention from the robotics community. On the theoretical side Suzuki and Yamashita first investigated the problem of a pursuer searching intruders using a beam sensor with unlimited range [14]. LaValle and colleagues further investigated this problem considering various restrictions and extensions. For example, the case of an omnidirectional unlimited range sensor was investigated [4], or the case of a robot equipped just with a *gap sensor*, i.e. a sensor capable only of detecting discontinuities [13]. On the more applied side, the formerly cited work on CMOMMT by Parker [11] set a milestone in the field. More recently Gerkey at al. [3] describe an implementation of the visibility based pursuit evasion problem on a robot with limited field of view.

Researchers in graph theory also investigated problems related to graph search. Three papers are particularly important in order to put our contribution into context. The concepts of *contaminated* and *clear* edges were introduced by Parsons [12], who pioneered this research vein. The problem he defined, called *edge-search*, deals with graphs where edges can be contaminated and have to be cleared by agents placed on vertices or marching along edges. The search number

$s(G)$ of the *edge-search* problem is the smallest number of agents with which one can find a sequence of actions, called *strategy*, such that all edges become clear. The problem of determining $s(G)$ was shown to be NP-hard by Megiddo et al. in [10]. An important extension to Parson's work was proposed by Barriere et al. [1], who first considered the edge-search problem with weighted vertices and edges. This extension implies that more than one agent is needed to perform the basic operations of clearing an edge or blocking a vertex. They also introduce the concept of contiguous strategies, i.e. solving strategies such that the clear subset of vertices always forms a connected subgraph of the original graph. They show that optimal contiguous strategies can be found in linear time on trees (contiguous strategies are however not optimal in general).

III. GRAPH CLEAR

This section offers a formalization of the GRAPH-CLEAR problem, pertinent notation and current algorithms for computing strategies on trees which serve as a basis for the new approach. Before moving to the formalism, we outline the connection between real world problems and the mathematical models presented herein. We are mainly interested in scenarios where robots operate in complex indoor environments with many rooms connected by multiple doors. In this scenario, rooms are modeled as graph vertices, while doors are mapped into graph edges connecting adjacent vertices (i.e. rooms). Figure 1 shows a simple environment and its corresponding graph model.

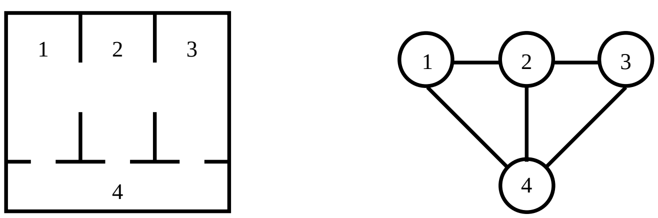

Fig. 1. An indoor environment and its corresponding graph model for the GRAPH-CLEAR problem.

As our focus is on robots with restricted capabilities, we assume that multiple robots are needed to patrol and search these environments. In particular, we suppose that in order to guarantee that no intruder crosses a door, we need to place a certain number of robots to guard it. This number is indicated as the *weight* of an edge. Similarly, more than one robot could be needed in order to sweep a room and make sure it contains no intruder, or detect them. This number is indicated as the *weight* of a vertex. While in the following we strictly stick to graph theory jargon, the reader could translate every instance of the word *vertex* with *room*, *edge* with *door* and *agent* with *robot*.

A. Definitions

GRAPH-CLEAR was formalized in [8] and is here shortly summarized. We define a *weighted graph*[1] as a triple $G = (V, E, w)$, where V is the set of vertices, E is the set of edges, and $w : V \cup E \to \mathbb{N} \setminus \{0\}$ is the weight function. The graph is undirected. Edges and vertices can be *clear* or *contaminated*. A clear vertex or edge hosts no intruders, while a contaminated vertex or edge could potentially hide one or more intruders. G is said to be clear when all vertices and edges are clear. A clear vertex v, however, can become contaminated again if there exists a path from v to another contaminated vertex or edge[2]. Recontamination of edges is analogue. Contaminated vertices or edges can be cleared by applying *clearing* and *blocking* operations respectively. A *clearing* operation applied to a vertex $v \in V$ detects all intruders under the assumption that no new intruders enter or leave the vertex. It hence clears the vertex using $w(v)$ agents. A *blocking* operation is applied to an edge $e \in E$ and detects all intruders passing through the edge and as such prevents contamination from spreading through the edge and clears it. The number of agents needed for a block is $w(e)$. When using multiple agents to clear a graph, we can deploy agents in edges or vertices in order to perform the blocking and clearing operations defined above. The policy we follow when deploying agents is called *strategy* and defined as:

*Definition 1 (*Strategy*):* Let $G = (V, E, w)$ be a weighted graph. A strategy S for G is a function $S : (V \cup E) \times \mathbb{N} \to \mathbb{N}$.

$S(x, t)$ is the number of agents deployed on $x \in V \cup E$ at time t. If $S(x, t) \geq w(x)$, then x is blocked or cleared at time t, depending on whether it is an edge or a vertex. Associated with each strategy there is a cost, i.e. the number of agents needed in order to implement the strategy.

*Definition 2 (*Cost of a strategy*):* Let $G = (V, E, w)$ be as before and let S be a strategy for G. The cost of S is

$$ag(S) = \max_{t \in \mathbb{N}} \sum_{x \in V \cup E} S(x, t) \quad (1)$$

A strategy S clears a graph G, if by deploying agents in the order dictated by the strategy, there exist a time t such that all edges and vertices are clear. While this notion could be made formal, we here omit the details. This leads us to the GRAPH-CLEAR problem.

*Definition 3 (*GRAPH-CLEAR problem*):* Let G be as before with all edges and vertices contaminated. Determine a strategy S for G that clears G and is of minimal cost $ag(S)$. The following formula gives the cost to clear a vertex *safely*, i.e. the cost to perform a clearing operation on the vertex while blocking all the edges connected to it to avoid immediate recontamination.

$$s(v) := w(v) + \sum_{e \in Edges(v)} w(e). \quad (2)$$

Any strategy which clears more than one vertex at time t has $ag(S) \geq ag(S')$ for some strategy $ag(S')$ which clears at most one vertex at time t. Therefore, as the focus of our research is in finding strategies of minimal cost, we will exclusively deal with strategies clearing at most one vertex

[1]while in graph related literature weighted graphs have weights for edges only, we instead assume that weights are defined both for edges and vertices.

[2]we also consider edges connecting two vertices in the path, contrary to the common definition of a path as a sequence of vertices. We opt not to formalize this slight difference to keep the notation simpler.

at once. There may be multiple strategies S of minimal cost, so we define the number of agents to clear a graph G as $ag(G) := ag(S)$ for any optimal strategy S.

B. Previous results for GRAPH-CLEAR

The concepts of contiguous and non-contiguous strategies play an important role in the algorithms we have formerly developed. As defined by Barriere at al. [1], a contiguous strategy requires that the subset of cleared vertices forms a connected subgraph. This requirement is relaxed for non-contiguous strategies. In [7] the GRAPH-CLEAR problem was first attacked, and an algorithm to produce non-contiguous strategies on trees was presented, as well as a upper bound on its cost w.r.t. to the depth of the tree. The algorithm is based on the computation of labels on edges which we will shortly present in this section. In [8] the NP-completeness of GRAPH-CLEAR was proven, and an algorithm to compute contiguous strategies on trees was presented. It was shown that both algorithms produce sub-optimal strategies for trees. The contiguous algorithm may, however, produce optimal contiguous strategies as discussed in [8]. Since contiguousness is a rather strict requirement that is not necessary in most robotics applications we investigate non-contiguous strategies to yield a lower number of agents. In [8] it was proposed to combine the two algorithms, i.e. the one producing sub-optimal non-contiguous strategies and the one producing contiguous strategies. An approach for finding non-contiguous strategies based on the two former algorithms, its theoretical properties and an improved algorithm are the primary contributions of this paper. First, we will shortly introduce the underlying mechanisms of the two previous algorithms. We restrict the problem to trees, and let $G_T = (V, E, w)$ be an instance of the GRAPH-CLEAR problem with G_T being a weighted tree. An instance of GRAPH-CLEAR on a graph can be reduced to an instance on a tree by permanently deploying a set of agents on suitable edges, so that the graph stays connected but exhibits no cycles. Since this cost is constant we will not consider it during the optimization process.

1) Non-contiguous labels: Let $v_x, v_y \in V$ and $e = [v_x, v_y] \in E$. We are assigning a label $\lambda_{v_x}(e)$ to edge e to represent the number of agents needed to clear the subtree rooted in v_y when entering from v_x. If v_y is a leaf, then $\lambda_{v_x}(e) = s(v_y) = w(v_y) + w(e)$. Otherwise consider all neighbors of v_y other than v_x. Let these be $v_2, \ldots, v_m$ with $m = degree(v_y)$. Write $e_i := [v_y, v_i]$ and let all v_i be ordered s.t. $\rho_i \geq \rho_{i+1}$ where $\rho_i := \lambda_{v_y}(e_i) - w(e_i)$. The ordering defines the sequence in which we clear the vertices v_i. The clearing cost of the subtree rooted at v_i is:

$$c(v_i) := \lambda_{v_y}(e_i) + \sum_{2 \leq l < i} w(e_l), \tag{3}$$

i.e. we have to use agents to block all edges to previously cleared subtrees and then use agents to clear the subtree rooted in v_i. The label on e hence becomes:

$$\lambda_{v_x}(e) = \max\{s(v_y), \max_{i=2,\ldots,m}\{c(v_i)\}\}. \tag{4}$$

The order defined by ρ_i minimizes this term. Once all labels are computed we can find a strategy to clear G_T from a vertex $v \in V$ with neighbors $v_1, \ldots, v_m$ by considering:

$$ag(v) = \max\left\{s(v), \max_{i=1,\ldots,m}\{c_{ag}(v_i)\}\right\}, \tag{5}$$

where $c_{ag}(v_i) = \lambda_v(e_i) + \sum_{1 \leq l < i} w(e_i)$ similar to $c(v_i)$, but including all neighbors since we do not enter from another vertex when we start the clearing from v directly. To find the minimal strategy we simply compute all labels and then select the vertex where $ag(v)$ is minimal. The resulting strategies are non-contiguous and not optimal. In fig. 2 the execution of a non-contiguous strategy based on the presented labels is illustrated. In [1] Barriere provides details for computing labels for a similar labeling mechanism in $O(n)$, where n is the number of vertices in the tree.

2) Contiguous labels: The contiguous variant of these labels is the basis of the contiguous algorithm. The key difference is that the contiguous strategy first clears v_y and then descends into the subtrees. It is motivated by the study of contiguous edge-search strategies for weighted trees by Barriere in [1]. Since we first clear v_y, all edges to vertices $v_2, \ldots, v_m$ have to remain blocked after safely clearing v_y. This means a reversal in the order in which we clear these vertices. Furthermore, when entering the subtree rooted in v_i we have the edge to v_i already blocked, contrary to the non-contiguous strategy. But the next step is to clear v_i itself before descending into the other subtrees. Figure 2 illustrates the difference between the contiguous and non-contiguous strategies. As we are using $s(v_i)$ agents for clearing v_i and also block e during this operation we can also take $\sum_{2 \leq l < i} w(e_l)$ as the additional number of agents to guard edges to contaminated neighbors rather than $\sum_{2 \leq l \leq i} w(e_l)$ as done in [8]. Once vertex v_i is cleared the block on e_i is removed and the term $\sum_{2 \leq l < i} w(e_l)$ remains the maximum number of agents used. Using this perspective it becomes apparent that contiguous and non-contiguous labels actually have the same equations complementing a lemma from [8] that the number of agent needed for a strategy based on non-contiguous labels is equal or better than contiguous labels and showing that the number of agents is indeed equal. In fig. 2 this becomes clearly visible and we therefore refrain from presenting a formal proof.

IV. HYBRID STRATEGIES

In [8] it was proposed to combine the two current algorithms by separating the neighboring vertices into two sets and clearing one using the contiguous and one with the non-contiguous algorithm. More precisely, for v_y, coming from v_x, we seek to partition the neighbors $V := \{v_2, \ldots, v_m\}$ into two sets of vertices V_1 and V_2. The first set V_1 will be cleared with the non-contiguous algorithm. Once all elements of V_1 are cleared the team clears v_y and then proceeds to clear V_2 with the contiguous algorithm. We thereby divide the weight of the term $\sum_{2 \leq l < i} w(e_l)$ from equation 3 onto two sets. This can greatly reduce the total number of agents

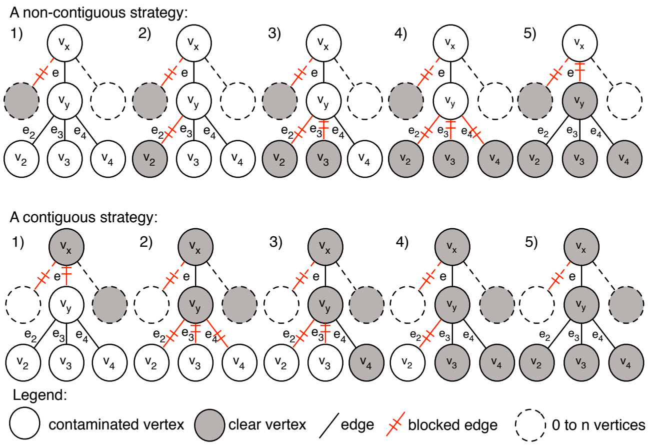

Fig. 2. Illustration of a contiguous and a non-contiguous strategy.

needed. Figure 3 illustrates how such a hybrid strategy would be executed.

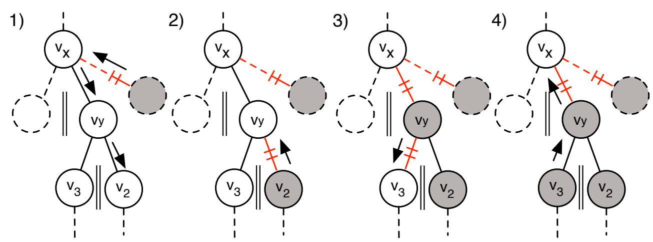

Fig. 3. Execution of the hybrid strategy.

From fig. 3 one complication becomes apparent. Let V_1^x and V_2^x be the partitioning of the neighbors of v_x when coming from yet another vertex v_z. If $v_y \in V_1^x$, then e is not blocked when the team enters v_y, as seen in fig. 3. Once we clear v_y we have to add a block on e which increases the total number of agents needed while clearing V_2, as seen in steps 3 to 5 in fig. 3. If $v_y \in V_2^x$, then the situation is reversed and we have to add a block on $w(e)$ only while we clear V_1 and not while clearing V_2.

Let us denote the case when $v \in V_1^x$ as case 1 and $v \in V_2^x$ as case 2. We can compute a label for both cases, using the superscripts ^{1} and ^{2}. So the labels on edge e become:

$$h_u^1(V_1, V_2) = \max\left\{\max_{v_i \in V_1}\{c^1(v_i)\}, \max_{v_i \in V_2}\{c^2(v_i) + w(e)\}\right\}$$

$$h_u^2(V_1, V_2) = \max\left\{\max_{v_i \in V_1}\{c^1(v_i) + w(e)\}, \max_{v_i \in V_2}\{c^2(v_i)\}\right\}$$

$$\lambda_{v_x}^1(e) = \max\left\{s(v_y), min_{V_1,V_2}\{h_u^1(V_1, V_2)\}\right\} \tag{6}$$

$$\lambda_{v_x}^2(e) = \max\left\{s(v_y), min_{V_1,V_2}\{h_u^2(V_1, V_2)\}\right\} \tag{7}$$

where $c(v_i)^j = \lambda_{v_y}^j(e_i) + \sum_{v_l \in V_j, 2 \le l < i} w(e_l)$ for $j = 1, 2$. It is easy to see, however, that $h_u^1(V_1, V_2) = h_u^2(V_2, V_1)$ given that $\lambda_{v_y}^1(e_i) = \lambda_{v_y}^2(e_i)$, which is the case since we compute the labels from the leaves upward and these equations are identical. It is however, important to note that the partition still has take into account the penalty term $w(e)$, i.e. only to which side it is assigned is not relevant. Hence, to simplify notation, we will drop superscripts ^{1} and ^{2}. The problem now states as follows:

Definition 4 (Hybrid algorithm: optimal partition): Given v_x, v_y and neighbors $V = \{v_2, \dots, v_m\}$ as before find a partition of V into V_1 and V_2 s.t. $h_u(V_1, V_2)$ is minimal.

The proposed algorithm to find partitions will be based on theoretical framework of the next two subsections. First we introduce the concept of batches which cluster vertices and then proceed by developing criteria for optimal partitions into V_1 and V_2 in section IV-B. On the basis of this we will develop an algorithm in section IV-C.

A. Batches

The following will be useful to describe at which vertex within a set V the number of agents is maximal. The proofs are left out here and can found in [**?**]. We shall call a set of all vertices with $\rho_i = a - p$ a batch B_p, where $a := \max\{\lambda_{v_y}(e_i)\}$. The set V can have at most $a - 1$ batches, i.e. $B_1, B_2, \dots, B_{a-1}$. During the execution of a strategy S in the non-contiguous variant we clear the batches in sequence $B_1, B_2, \dots, B_{a-1}$ and then clear v. For the contiguous variant the order of clearing is reversed. Define the weight of a batch as $w(B_p) := \sum_{v_i \in B_p} w(e_i)$ and write $w(B_{p<k}) := \sum_{p<k} w(B_p)$. Define the maximum cost within V to be $h := \max_{2 \le i \le m}\{c(v_i)\}$ and let v_q be a vertex that assumes this maximum, i.e. $h = c(v_q)$, s.t. $v_q \in B_k$ with k being the largest such possible batch index. Using this notation we can rewrite the maximum cost to be:

$$\begin{aligned} h &= w(B_{i<k}) + w(B_k) - w(e_q) + \lambda(e_q) \\ &= w(B_{i \le k}) + \rho_q = w(B_{i \le k}) + a - k. \end{aligned} \tag{8}$$

The following lemma will be relevant for our further results.

Lemma 1: Let v_q and B_k be as before. Consider any non-empty $B_{k'}$ s.t. $k \neq k'$. If $k > k'$, then $k - k' \le w(B_{k'<i\le k})$. Otherwise if $k < k'$, then $w(B_{k<i\le k'}) \le k' - k$.

B. Criteria for optimal partitions

All vertices in batches B_i, for $i > k$, do not contribute to the maximum, i.e. a removal of these vertices does not change the maximum cost. We shall call such vertices the *tail* $T := \bigcup_{i>k} B_i$ of V. Their joint weight shall be denoted by $w(T) = \sum_{v_i \in T} w(e_i)$. As a consequence of lemma 1 we have $w(T_t) < a - k$.

When partitioning V into V_1 and V_2 we shall write $B_{i,1}$, $B_{i,2}$ for the batches of V_1 and V_2, k_1, k_2 for k, $v_{q,1}$, $v_{q,2}$ for v_q, h_{V_1}, h_{V_2} for h and T_1 and T_2 for T. For notational simplicity we will ignore the penalty term in this section and discuss it thereafter when presenting the partitioning algorithm. Finally, for a partition V_1 and V_2 we define a maximization criterion as:

$$c(V_1, V_2) := k_1 + k_2 + w(T_1) + w(T_2) - |h_1 - h_2|. \tag{9}$$

Definition 5 (Balanced and full partitions): Let V be a set of vertices as before. A partitioning of V into V_1 and V_2 is called

- *full* if $k = k_1 = k_2$,
- *balanced* if $w(B_{i\leq k_1,1}) - k_1 = w(B_{i\leq k_2,2}) - k_2$,
- *maximal* if for any other partition V_1', V_2' we get that $c(V_1, V_2) \geq c(V_1', V_2')$.

It is easy to see that a partition that is full and balanced will minimize h_u and is therefore optimal. Also any full and balanced partition will be maximal. To show that any maximal partition is optimal we need the following lemma to show that $h_b := w(B_{i\leq k})/2 + a - k$ is a lower bound on h_u.

Lemma 2: Given V, with a and k as before and any partition V_1 and V_2 we have that:

$$h_u \geq w(B_{i\leq k})/2 + a - k = h_b. \tag{10}$$

Proof: Detailed proof found in [9].

For full and balanced partitions we have $h_u = h_b$. But a full and balanced partition may not exist and hence we have to consider maximal partitions.

Lemma 3: If V_1, V_2 is a maximal partition of V, then h_u is minimal, i.e. the partition is optimal.

Proof: Detailed proof found in [9].

In colloquial terms, we have to find a partition with the largest k_1, k_2 and large tails T_1, T_2 and with $w(B_{i\leq k_1,1})$ roughly equal to $w(B_{i\leq k_2,2})$.

C. The partitioning algorithm

The algorithm is based on a dynamic programming approach motivated by the relation of the maximization criterion to the subset sum problem, one of the early NP-complete problems [2]. In short, the subset sum problem is to determine whether a set of integer values contains a subset whose values sum up to some given integer z. A dynamic programming algorithm to solve it runs in pseudo-polynomial time $O(Cn)$ where C is the sum of all members of the set and n is the number of elements. Translated to our case this becomes the problem to determine whether V contains a set of vertices V_2 s.t. the sum of the weight of their respective edges $w(V_2)$ sums up to $z = \lceil w(V)/2 - w(e)/2\rceil$. Here $w(e)$ is the penalty term from equation 6. A solution V_2 would minimize h_u given that $V_1 = V \setminus V_2, V_2$ is a full partition, i.e. it satisfies $k_1 = k_2 = k$. Obviously, using the dynamic programming approach for solving the subset sum problem gives no guarantee that $k_1 = k_2 = k$. In fact, such a partition may not even exist. The following will be concerned with an algorithm that guarantees to find a full partition if one exists.

Let A be a table with $m-1$ rows and $z = \lceil w(V)/2 - w(e)/2\rceil$ columns. Set $A(0, j) := 0, \forall j$ and $A(i, 0) := 0, \forall i$. Each row represents a vertex and they shall be ordered as $v_m, \dots, v_2$, i.e. v_m corresponds to row one, v_{m-1} to row two and so on. Write c_i for $w(e_{m-i+1})$, i.e. the cost added to V_2 by adding the vertex in row i. If $c_i > j$, then $A(i, j) = A(i-1, j)$, otherwise $A(i, j) = \max\{A(i-1, j), A(i-1, j-c_i) + c_i)\}$. An entry $A(i, j)$ in the table is then the maximal weight for V_2 achievable using vertices $v_m, \dots, v_{m-i+1}$. The table is filled as usual for the subset sum problem. If an entry in A exists s.t. $A(i, j) = \lceil w(V)/2 - w(e)/2\rceil$, then we have a partition that is optimal w.r.t. to the distribution of the edge weights onto V_1 and V_2. This is, however, only one part in the optimization. An entry in A represents possibly multiple partitions, some of which do not satisfy that $k_1 = k_2 = k$. A particular partition can be thought of as a path within the table. In [9] examples illustrating this are presented. Finding an optimal partition is hence the problem of finding an entry with $A(i, j) = \lceil w(V)/2 - w(e)/2\rceil$ for which we have a path that represents a maximal partition. We will show how to compute whether such a path exists for the case of full partitions.

Since we ordered the vertices in reverse order we can view the problem from the perspective of adding vertex by vertex with decreasing index to V_2 as we proceed through the rows of A. For V_1 we can view it as if we are removing vertices with decreasing index from V_1. The main question is what happens to k_1 for V_1 and k_2 for V_2 as we remove and add vertices. When we add a vertex $v \in B_{u,1}$ from V_1 to V_2 we know that all other vertices in V_2 are in batches $B_{j\geq u,2}$. Write $V_1' = V_1 \setminus \{v\}$ and $V_2' = V_2 \cup \{v\}$. Define $S(V_2) := \sum_{1\leq i\leq k_2} w(B_i, 2)$ to be the support of V_2. Now if $k_2 - u > S(V_2)$, then $v = v_q'$ will be the new maximum for V_2'. Otherwise, if $k_2 - u \leq S(V_2)$, then $v_q = v_q'$. To illustrate this with our example set of vertices simply choose $V_2 = \{v_9\}$. Clearly $v_{q,2} = v_9$ and $S(V_2) = 2$ and adding v_5 will lead to $v_{q,2}' = v_5$. Similarly for V_1', when removing v with associated edge e_v, the support will be reduced to $S(V_1') = S(V_1) - w(e_v)$. Now the maximum $v_{q,1}'$ may shift to a vertex of a lower batch if $\exists B_b$ s.t. $k_1 - b > S(V_1)$, otherwise it will remain at it former vertex s.t. $v_{q,1}' = v_{q,1}$.

As long as $k_1 = k_2 = k$ we know that $S(V_1) = w(V_1), S(V_2) = w(V_2), w(T_1) = w(T_2) = 0$ and we do not need to keep track of these values. Once we add a vertex $v \in B_{u,1}$ from V_1 to V_2 with $k_2 - u > S(V_2)$ we will have $k_2' < k$ and the path will not be a valid solution. Let us define two further tables $K_1(i, j)$ and $K_2(i, j)$ in which we will keep track of k_1 and k_2. For our case the computation of $K_1(i, j)$ and $K_2(i, j)$ involves only a simple check, whether upon addition/removal of the vertex the current K_1 and K_2 can be maintained. If this is not the case we discard the solution path by setting $K_1(i, j) = 0$ or $K_2(i, j) = 0$. The pseudo code in 1 shows how to compute A, K_1 and K_2. Initially we set $K_1(0, j) = K_2(0, j) = k$. It is obvious that k_1, k_2 are monotonically decreasing w.r.t. to growing i, j, except for the special case for V_1 if we remove the first vertex v_2 in the last row of the table and at this point have $v_{q,1} = v_2$ and $B_{b_1,1} = \{v_2\}$, i.e. there is no other vertex in its batch. Dealing with this special case merely complicates notation without changing the methodology and we will therefore ignore it. Now, an entry $A(i, j) = z$ with $K_1(i, j) = K_2(i, j) = k$ has a path that represents a full and balanced partition which is therefore optimal. If no such entry exists, then neither does a full and balanced partition. In [9] a starting point to construct an analogue algorithm for

Algorithm 1 $Compute\_table\_entry(i, j)$

if $c_i > j$ **then**
  $A(i, j) \leftarrow A(i-1, j)$
  $K_1(i, j) \leftarrow K_1(i-1, j)$
  $K_2(i, j) \leftarrow K_2(i-1, j)$
else
  $A(i, j) = \max\{A(i-1, j), A(i-1, j-c_i) + c_i)\}$
  if $A(i, j) = A(i-1, j-c_i) + c_i$ **then**
    if $\rho_2 < K_1(i-1, j-c_i) - (w(V) - A(i, j))$ **then**
      $K_1(i, j) \leftarrow 0$
    else
      $K_1(i, j) \leftarrow K_1(i-1, j-c_i)$
    end if
    if $a - \rho_{m-i} < K_2(i-1, j-c_i) - A(i-1, j-c_i)$ **then**
      $K_2(i, j) \leftarrow 0$
    else
      $K_2(i, j) \leftarrow K_2(i-1, j-c_i)$
    end if
  end if
  if $A(i, j) = A(i-1, j)$ **then**
    if $K_2(i-1, j) \geq K_2(i, j)$ and $K_1(i-1, j) \geq K_1(i, j)$ **then**
      $K_1(i, j) \leftarrow K_1(i-1, j)$
      $K_2(i, j) \leftarrow K_2(i-1, j)$
    end if
  end if
end if

maximal partitions is given.

V. Discussion and Conclusion

We presented a new approach for finding strategies for GRAPH-CLEAR in a tree which requires solving a partitioning problem. We presented criteria for optimal partitions and based on these we presented an algorithm that computes optimal partitions given that a full and balanced partition exist. In [9] the open problem to develop to compute maximal partitions is discussed in more detail. Apart from various other details dealt with in [9], the key problem for finding maximal partitions efficiently is to find a way to compute $C(i, j)$ from the entries for A, K_1, K_2, T_1, T_2 at $(i-1, j)$ and $(i-1, j-c_i)$. We do already know how K_1, K_2, T_1 and T_2 evolve when adding a vertex v_{m-i+1}. Hence we can identify whether the path from $(i-1, j)$ or from $(i-1, j-c_i)$ leads to a better partition w.r.t. to $C(i, j)$. The only open problem that remains is whether a path representing an optimal partition at entry $A(i, j)$ will be optimal at all previous entries. In other words, it is the question whether the partition for $A(i, j)$ that maximizes $C(i, j)$ can be computed based on the description of the partitions in the previous row $i-1$.

Even without computing maximal partitions a greedy partition for a hybrid method already outperforms the previous algorithms from [7] and [8]. On the other hand, in a realistic robotic application of GRAPH-CLEAR a reduction by a few robots is already significant decrease in costs and therefore an optimal solution for the partitioning is of interest. Furthermore, from a graph-theoretical perspective the investigation whether optimal solutions for general strategies on trees exist motivates further analysis of the hybrid method. Yet, we believe the current progress for finding GRAPH-CLEAR strategies already gives a good basis for using it in robotic applications for surveillance.

References

[1] L. Barriere, P. Flocchini, P. Fraigniaud, and N. Santoro. Capture of and intruder by mobile agents. In *Proceedings of the 14th annual ACM symposium on Parallel algorithms and architectures*, pages 200–209, 2002.

[2] M.R Garey and D.S. Johnson. *Computers and Intractability. A guide to the theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[3] B. Gerkey, S. Thrun, and G. Gordon. Visibility-based pursuit-evasion with limited field of view. *International Journal of Robotics Research*, 25(4):299–316, 2006.

[4] L.J. Guibas, J.-C. Latombe, S.M. LaValle, D. Lin, and R. Motwani. A visibility-based pursuit-evasion problem. *International Journal of Computational Geometry and Applications*, 9(4/5):471–494, 1999.

[5] A. Kolling and S. Carpin. Multirobot cooperation for surveillance of multiple moving targets - a new behavioral approach. In *Proceeding of the IEEE International Conference on Robotics and Automation*, pages 1311–1316, 2006.

[6] A. Kolling and S. Carpin. Cooperative observation of multiple moving targets: an algorithm and its formalization. *International Journal of Robotics Research*, 26(9):935–953, 2007.

[7] A. Kolling and S. Carpin. Detecting intruders in complex environments with limited range mobile sensors. In K. Kowzlowski, editor, *Robot Motion and Control 2007*, Lecture Notes in Information and Control, pages 417–426. Springer, 2007.

[8] A. Kolling and S. Carpin. The graph-clear problem: definition, theoretical properties and its connections to multirobot aided surveillance. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1003–1008, 2007.

[9] A. Kolling and S. Carpin. An improved algorithm for the GRAPH-SEARCH problem. Technical Report 2007-3, UC Merced, 2007.

[10] N. Megiddo, S.L. Hakimi, M.R. Garey, D.S. Johnson, and C.H. Papadimitriou. The complexity of searching a graph. *Journal of the ACM*, 35(1):18–44, 1988.

[11] L. E. Parker. Distributed algorithms for multi-robot observation of multiple moving targets. *Autonomous robots*, 12(3):231–255, 2002.

[12] T. Parsons. Pursuit-evasion in a graph. In Y. Alavi and D. Lick, editors, *Theory and Applications of Graphs*. Springer-Verlag, 1976.

[13] S. Sachs, S. Rajko, and S. M. LaValle. Visibility-based pursuit-evasion in an unknown planar environment. *International Journal of Robotics Research*, 23(1):3–26, 2004.

[14] I. Suzuki and M. Yamashita. Searching for a mobile intruder in a polygonal region. *SIAM Journal of Computing*, 21(5):863–888, 1992.