An improved algorithm for the GRAPH-CLEAR problem.

Andreas Kolling and Stefano Carpin

University of California
School of Engineering
Merced,CA, USA

Summary. The main contribution of this paper are improved algorithms 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 algorithms combine 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. This is followed by the presentation of an algorithm that finds an optimal solution under these special circumstance and a second algorithm that attempts to generalize the previous one. Finally we also identify a set of open questions deserving more investigations.

1 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 [10]. 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 2 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 3 summarizes the GRAPH-CLEAR problem and shortly addresses our formerly developed algorithms. The new approach and a theoretical framework are presented in section 4, followed by a new algorithm that computes strategies for the new approach under certain conditions and an attempt to generalize the presented algorithm to for general strategies. Section 5 concludes with a discussion of remaining problems and possible extensions of the presented work.

2 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 [13]. 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 [12]. On the more applied side, the formerly cited work on CMOMMT by Parker [10] 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 [11], 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 [9]. 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).

3 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 a new solving 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.

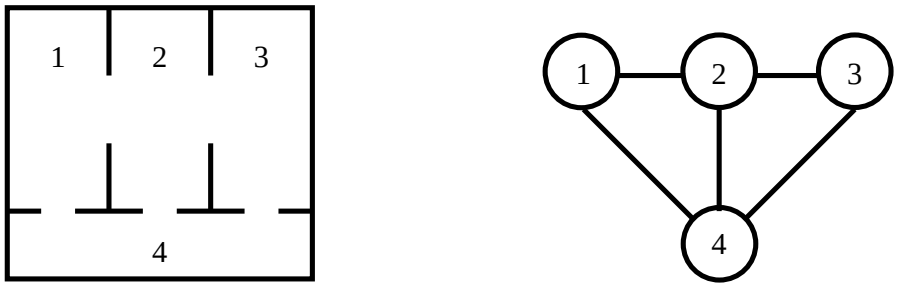

Fig. 1. An indoor environment and its corresponding graph model for the GRAPH-CLEAR problem.

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*.

3.1 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:

1. *Clearing* - an action applied to a vertex $v \in V$ that ensures that all intruders are detected, assuming no new intruders enter or leave the vertex through an edge that is not blocked. When this operation is applied the vertex becomes clear. The number of agents needed for clearing is $w(v)$.
2. *Blocking* - an action applied to an edge $e \in E$ that does not allow recontamination of any edge or vertex through a path using the blocked edge. A blocked edge becomes clear. The number of agents need for a block is $w(e)$.

When using multiple agents in order to clear a graph, we can deploy agents in edges or vertices in order to perform the blocking and clearing actions 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. Associated with each strategy there is a cost, i.e. the number of agents needed in order to implement the strategy.

[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.

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

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

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 $G = (V, E, w)$ be a weighted graph 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.

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

The definition allow us to place agents on multiple vertices in one step. It is worth noting that for 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. Furthermore, for a graph G there may be multiple strategies S of minimal cost, so we define the number of agents need to clear a graph G as $ag(G) := ag(S)$ for any optimal strategy S for G.

3.2 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 lower bound on the number of agents w.r.t to the depth of the tree. The algorithm is based on the computation of labels on edges which we will quickly present in this section. In [8] 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 mentioned in the discussion in [8]. In [1] contiguous strategies on weighted graphs for the edge-search problem were first studied. Therein it was shown that for edge-search a similar labeling method than used in [8] actually produces optimal strategies. The problems GRAPH-CLEAR and

edge-search are however different, so although similarities exist, it is not clear if this result can be safely extended to the GRAPH-CLEAR problem. 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 needed. 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. In the following 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 of GRAPH-CLEAR 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, it will not be mentioned anymore from now on, and it will not play a role in the optimization process.

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 $m = \mathit{degree}(v_y)$. 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. Now define the clearing cost of clearing the subtree rooted at v_i as:

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

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 the tree 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, upper parts, 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 $O(n)$ time, where n is the number of vertices in the tree.

Contiguous labels

The contiguous variant of these labels is the basis of the contiguous algorithm from [8]. 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. At first sight this requires a modification of the costs and computation of labels as presented in [8]. But for this paper let us consider a simplification. When entering v_i edge e_i is blocked. 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.

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

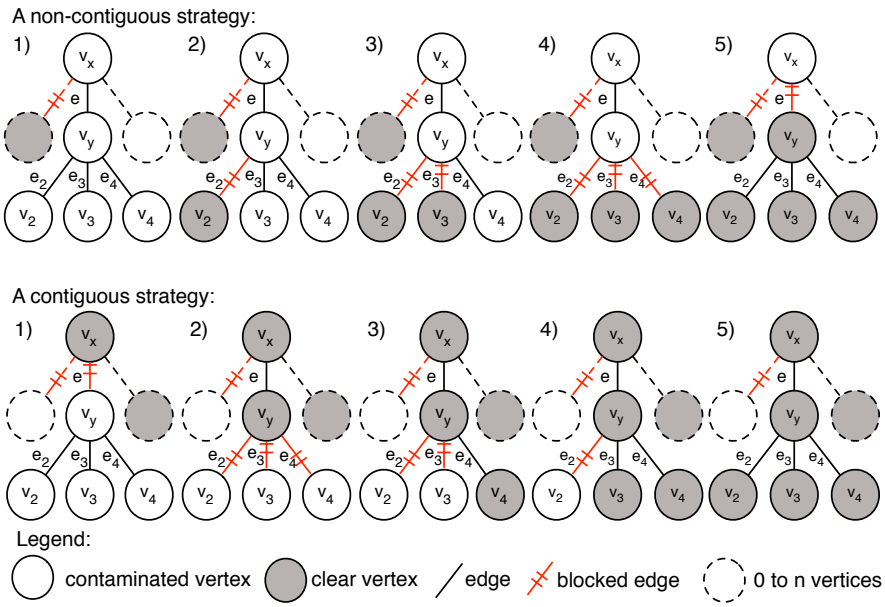

Fig. 2. An illustration of the execution of a strategy produced by the non-contiguous and the contiguous algorithms. In step one for the non-contiguous strategy the team descends into the subtree at v_y and then proceeds by clearing v_2 in step 2 using $s(v_2)$ agents. Once cleared the edge to v_2 has to remain blocked. This procedure repeat in steps 3 and 4 for v_3 and v_4. Once all subtrees are cleared the team clears v_y with $s(v_y)$ and then releases the blocks on e_2, e_3 and e_4 and adding a block to e. The maximum number of agents located at any time within the subtree at v_y becomes the label λ_{v_y} for e. For the contiguous strategy the team first clears v_y and blocks the edge to all subtrees. It then clears the subtrees in reverse order than for the non-contiguous version, removing the blocks for each edge as the subtree becomes cleared.

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:

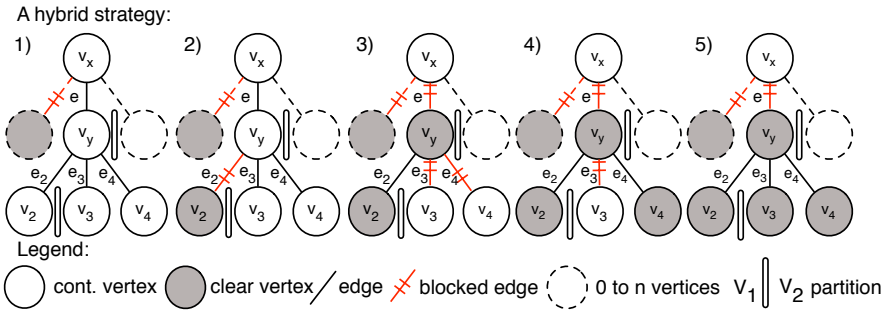

Fig. 3. Execution of the hybrid strategy. At step 1 the robot team enters the subtree rooted at v_y and proceeds by clearing $V_1 = \{v_2\}$. In step 2 v_2 is cleared and the edge to $[v_y, v_2]$ is blocked and the team proceeds to clear v_y. Then in step 3 and 4 it clears $V_2 = \{v_3, v_4\}$. Once done the team returns to v_x and continues clearing the neighbors of v_y.

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

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

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

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

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 $V = \{v_2, \ldots, v_m\}$ 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 certain types of partitions satisfying that $h_u(V_1, V_2)$ is minimal will be based on theoretical framework of the next two subsections. First we introduce the concept of batches which cluster vertices in section 4.1 and then proceed by developing criteria for optimal partitions into V_1 and V_2 in section 4.2. On the basis of this we will develop the actual algorithm in section 4.3.

4.1 Batches

The following definition will be useful to describe occurrences of the maximum number of agents within a set of vertices V when clearing it with either the contiguous or non-contiguous algorithm.

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. $B_1, B_2, \ldots, B_{a-1}$. During the execution of a strategy S in the non-contiguous variant we clear the batches in sequence $B_1, B_2, \ldots, B_{a-1}$ 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 $h := \max_{2 \leq i \leq m}\{c(v_i)\}$ 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:

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

Furthermore, we can define the maximum cost within a batch:

$$h_j := \begin{cases} w(B_{i \leq j}) + a - j & if B_j \neq \emptyset \\ 0 & otherwise \end{cases} \tag{9}$$

Clearly $h = \max_{1 \leq j \leq a-1}\{h_j\}$.

There are some simple results with regard to the weight of batches and their distance to the current B_k within V.

Lemma 1. *Let v_q and B_k be as before. Consider any non-empty batch $B_{k'}$ s.t. $k > k'$. Then*

$$k - k' \leq w(B_{k'<i \leq k}). \tag{10}$$

Proof: Given h as above, consider the last vertex v_r of another batch $B_{k'}$, i.e. $v_r = v^e_{k'}$ and define $h' := c(v_r) = w(B_{i \leq k'}) + \rho_r$. Recall that $\rho_q = a - k$ and $\rho_r = a - k'$. By assumption $h' \leq h$. This implies

$$\begin{aligned} w(B_{i \leq k'}) + \rho_r &\leq w(B_{i \leq k}) + \rho_q \\ w(B_{i \leq k'}) + \rho_r - \rho_q &\leq w(B_{i \leq k}) \\ k - k' &\leq w(B_{i \leq k}) - w(B_{i \leq k'}) \\ k - k' &\leq w(B_{k'<i \leq k}) \end{aligned} \tag{11}$$

which concludes the proof.□ An analogue result exists for $k < k'$.

Lemma 2. *Let v_q and B_k be as before. Consider any non-empty batch $B_{k'}$ s.t. $k < k'$. Then*

$$w(B_{k<i \leq k'}) \leq k' - k. \tag{12}$$

Proof: Analogue to proof of lemma 1.□
We can list all classes of vertices in a batch B_p as:

$$\begin{array}{l|cccc|} \rho_i & a-p & a-p & \dots & a-p \\ \lambda(e_i) & a-p+1 & a-p+2 & \dots & a \\ w(e_i) & 1 & 2 & \dots & p \end{array}$$

Using this we can write down all batches and their possible edge classes as:

$$\begin{array}{l|c|cc|ccc|} Batch & B_1 & B_2 & & \dots & B_{a-1} & \\ \rho_i & a-1 & a-2 & a-2 & \dots & 1\ 1\ \dots & 1 \\ \lambda(e_i) & a & a-1 & a & \dots & 2\ 3\ \dots & a \\ w(e_i) & 1 & 1 & 2 & \dots & 1\ 2\ \dots & a-1 \end{array}$$

Table 1 shows a set of vertices $V = \{v_2, v_3, \dots, v_{10}\}$ with $a = 10$ and $v_q = v_9$ with maximum cost $c(v_q) = 18$.

B_p	B_2	B_3			B_5	B_6	B_7		B_9
v_i	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}
ρ_i	8	7	7	7	5	4	3	3	1
$\lambda(e_i)$	10	8	8	10	7	7	5	5	2
$w(e_i)$	2	1	1	3	2	3	2	2	1
$c(v_i)$	10	10	11	14	14	15	16	18	17

Table 1. A simple example of a set of vertices and their assignment into batches.

4.2 Criteria for optimal partitions

This section discusses criteria for optimal partitions. As seen from the example set from Table 1 there are vertices in batches $B_i, i > k$ that do not contribute to the maximum such as vertex $v_{10} \in B_9$ in the example. We shall call all 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 $B_{i,1}$ and $B_{i,2}$ for the batches of V_1 and V_2 respectively. Furthermore, we write k_1 and k_2 for previously k, $v_{q,1}$ and $v_{q,2}$ for previously v_q and h_{V_1} and h_{V_2} for previously h. Also the tails become T_1 and T_2. For notational simplicity we will also ignore the penalty term formerly discussed as this does not change the basic results. It will be introduced again when presenting the partitioning algorithm. Finally, for a partition V_1 and V_2 define its maximization criteria as:

$$c(V_1, V_2) := k_1 + k_2 + w(T_1) + w(T_2) - |h_1 - h_2|. \tag{13}$$

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 3. *Given* V*, with* a *and* k *as before and any partition* V_1 *and* V_2 *we have that:*

$$h_u \geq w(B_{i\leq k})/2 + a - k = h_b. \tag{14}$$

Proof: W.l.o.g. assume that $h_{V_1} \geq h_{V_2}$, i.e. $w(B_{i\leq k_1,1}) + a - k_1 \geq w(B_{i\leq k_2,2}) + a - k_2$. So $h_u = h_{V_1}$. Since by assumption V has no tail we have that $w(T_1) \leq k - k_1$ and $w(T_2) \leq k - k_2$. Assume that $h_{V_1} < w(B_{i\leq k})/2 + a - k$ which leads to:

$$\begin{aligned} 2 \cdot h_{V_1} &< w(B_{i\leq k_1,1}) + w(B_{i\leq k_2,2}) && (15)\\ &\quad w(T_1) + w(T_2) + 2a - 2k \\ 2 \cdot h_{V_1} &< w(B_{i\leq k_1,1}) + w(B_{i\leq k_2,2}) && (16)\\ &\quad a - k_1 + a - k_2 \\ h_{V_1} &< h_{V_2} && (17) \end{aligned}$$

Which is a contradiction to $h_{V_1} \geq h_{V_2}$ and concludes the proof.□

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

Lemma 4. *If* V_1, V_2 *is a maximal partition of* V*, then* h_u *is minimal.*

Proof: W.l.o.g. assume that $k_2 \leq k_1$. As before we have $w(B_{i\leq k}) = w(B_{i\leq k_1,1}) + w(T_1) + w(B_{i\leq k_2,2}) + w(T_2)$ and $|h_1 - h_2| = |(w(B_{i\leq k_1,1}) + k_1) - (w(B_{i\leq k_2,2}) + k_2)|$. This leads to the following cases:

Case 1: assume $h_u = h_1 > h_2$ and we get

$$h_2 - h_1 + k_2 - k_1 = w(B_{i\leq k_2,2}) - w(B_{i\leq k_1,1}). \tag{18}$$

Now:

$$\begin{aligned} h_b - h_u &= w(B_{i\leq k})/2 - w(B_{i\leq k_1,1}) + k_1 - k \\ &= \frac{1}{2}\left(h_2 - h_1 + k_2 + k_1 + w(T_2) + w(T_1)\right) - k \end{aligned} \tag{19}$$

By the maximal property we get that $h_b - h_1$ is maximal and the partition is therefore optimal. Case 2: assume $h_u = h_2 > h_1$ and analogue to the previous case this results in:

$$h_b - h_2 = \frac{1}{2}\left(h_1 - h_2 + k_2 + k_1 + w(T_2) + w(T_2)\right) - k \tag{20}$$

which is again maximal by the maximal property of the partition. Hence a maximal partition is optimal.□

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 $w(B_{i\leq k_2,2})$.

4.3 The partitioning algorithm

The results of the previous section give us a good starting point for the algorithms in this section. They are based on a dynamic programming approach motivated by the relation 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 and balanced partition if one exists.

Now, 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 $v_m, v_{m-1}, \ldots, v_2$, 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 $v_m, \ldots, v_{m-i+1}$. In table 4 these values are computed for our simple example from table 1. As mentioned before we can assume that V has no tail and hence ignore the tail $\{v_{10}\}$ of the example and therefore have only 8 vertices. For the example table 4 we assume that $w(e) = 2$, which means we seek to create V_2 with $w(V_2) = 7$.

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. The weight of V_1 and V_2 is, however, only one part in the optimization. In table 4 each entry with $A(i, j) = 7$ represents possibly multiple partitions, some of which do not satisfy that $k_1 = k_2 = k = 7$. In figure 5 three possible partitions, represented as paths through the table, with $w(V_2) = 7$ are shown with $V_2 = \{v_9, v_7, v_4, v_3\}$ being the only optimal one. The other two lead to $k_1 = 5$ or $k_1 = 3$ and are therefore not optimal. 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

Column			1	2	3	4	5	6	7
v_i	$w(e_i)$	Row							
v_9	2	1	0	2	2	2	2	2	2
v_8	2	2	0	2	3	4	4	4	4
v_7	3	3	0	2	3	4	5	6	7
v_6	2	4	0	2	3	4	5	6	7
v_5	3	5	0	2	3	4	5	6	7
v_4	1	6	1	2	3	4	5	6	7
v_3	1	7	1	2	3	4	5	6	7
v_2	2	8	1	2	3	4	5	6	7

Fig. 4. The dynamic programming table for the example from table 1 to solve the subset sum problem.

which we have a path that represents a maximal partition. We will illustrate how to compute whether such a path exists for the case of full and balanced partitions. To do so we need to find an efficient way to keep track of the possible k_1, k_2 that one can have when at $A(i, j)$.

Column			1	2	3	4	5	6	7
v_i	$w(e_i)$	Row							
v_9	2	1	0	2	2	2	2	2	2
v_8	2	2	0	2	3	4	4	4	4
v_7	3	3	0	2	3	4	5	6	7
v_6	2	4	0	2	3	4	5	6	7
v_5	3	5	0	2	3	4	5	6	7
v_4	1	6	1	2	3	4	5	6	7
v_3	1	7	1	2	3	4	5	6	7
v_2	2	8	1	2	3	4	5	6	7

Fig. 5. Possible partitions resulting from the dynamic programming table. A partition is represented by a sequence of arrows where a diagonal arrow means that the vertex of the row to which the arrow is pointing is in V_2 while a horizontal arrow indicates that the vertex is in V_1. A valid partition consists only of arrows with dark tips while invalid partitions have at least one arrow with a hollow tip.

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.

Now 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 $\exists B_b \neq \emptyset$ s.t. $k_1 - b > S(V_1)$, otherwise it will remain at its 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 as they evolve. 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 since k_2 can only grow upon addition of the first vertex. Let us define two further tables $K_1(i,j)$ and $K_2(i,j)$ for the dynamic programming approach 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 or 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 and it was the maximum in V_2 before removal. Dealing with this special case merely complicates notation without changing the methodology and we will therefore ignore it for now.

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. The algorithm for this case of full and balanced partitions illustrates how to use the theoretical results of this paper to obtain an improved algorithm for the GRAPH-CLEAR problem on trees.

Trying to obtain an algorithm for the general case entails more complications. In essence, a compromise between obtaining balanced edge weight and large k_1, k_2 has to be sought. More precisely, we have to consider all parts of the maximization criteria $c(V_1, V_2)$ to identify optimal partitions. Extending the previous dynamic programming approach with brute force would mean to evaluate all possible paths in the table leading to any entry $A(i,j)$ and choosing one for which the maximization criteria is largest. Obviously, this is not efficient. To see how we could arrive at a more efficient method let us define $C(i,j)$ to be the largest value of $c(V_1, V_2)$ across all partitions that lead to a path to $A(i,j)$. More precisely, $C(i,j)$ is the largest $c(V_1, V_2)$ for all partitions V_1, V_2 s.t. $V_2 \subset \{v_m, \ldots, v_{m-i+1}\}$ with $w(V_2) = A(i,j)$. Computing C involves keeping track not only of k_1 and k_2 in tables K_1, K_2, but also of the tails T_1 and T_2 for whose weight we also need tables $T_1(i,j)$ and $T_2(i,j)$ and requires therefore some more bookkeeping. Using equation 13 for $c(V_1, V_2)$ we get that:

Algorithm 1 *Compute_table_entries_at*(i, j)

```
if c_i > j then
  A(i,j) ← A(i−1,j)
  K1(i,j) ← K1(i−1,j)
  K2(i,j) ← K2(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 ρ2 < K1(i−1,j−c_i) − (w(V) − A(i,j)) then
      K1(i,j) ← 0
    else
      K1(i,j) ← K1(i−1,j−c_i)
    end if
    if a − ρ_{m−i} < K2(i−1,j−c_i) − A(i−1,j−c_i) then
      K2(i,j) ← 0
    else
      K2(i,j) ← K2(i−1,j−c_i)
    end if
  end if
  if A(i,j) = A(i−1,j) then
    if K2(i−1,j) ≥ K2(i,j) and K1(i−1,j) ≥ K1(i,j) then
      K1(i,j) ← K1(i−1,j)
      K2(i,j) ← K2(i−1,j)
    end if
  end if
end if
```

$$C(i,j) = K_1(i,j) + K_2(i,j) + T_1(i,j) + T_2(i,j) \tag{21}$$

$$- |H_1(i,j) - H_2(i,j) - w(e)| \tag{22}$$

$$H_1(i,j) = w(V) - A(i,j) - T_1(i,j) - K_1(i,j), \tag{23}$$

$$H_2(i,j) = A(i,j) - T_2(i,j) - K_2(i,j). \tag{24}$$

where K_1, K_2, T_2, T_1 now describe a partition that minimizes $C(i,j)$. Since the support of a set V_1 is s.t. $w(V_1) = S(V_1) + w(T_1)$ we also know about the size of the support of V_1 and analogue for V_2.

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 $C(i,j)$. One problem, however, is that it not known whether it is possible that a partition at $(i-1,j)$ or $(i-1, j-c_i)$ leads to an optimal partition at (i,j) while not being optimal for $C(i-1,j)$ or $C(i-1, j-c_i)$ respectively.

To see this more clearly consider a partition for $A(i,j)$ that maximizes $C(i,j)$. Now the vertex for row i, i.e. v_{m-i+1}, is either in V_2 or V_1. Now if $v_{m-i+1} \in V_2$ for this partition, then the partition $V_2' = V_2 \setminus \{v_{m-i+1}\}, V_1' =$

$V \setminus V_2'$ is a partition that leads to a path to $A(i-1, j-c_i)$. There is no proof yet that this partition V_2', V_1' maximizes $C(i-1, j-c_i)$. If this is not the case, then a path leading to a maximal partition for (i, j) passing through $(i-1, j-c_i)$ can be different from the optimal path to entry $(i-1, j-c_i)$. This has dramatic consequences, as we cannot build our solution path row by row but would have to reconsider all possible paths to an entry. Let us illustrate this with an example in table 2 and 6. Table 7 shows some partitions represented by paths that lead to entry $A(5, 4)$. These three partitions from [**?**] $V_2' = \{v_3, v_6\}$, $V_2'' = \{v_3, v_4, v_5\}$ and $V_3''' = \{v_3, v_4, v_7\}$. If we assume that the penalty term $w(e) = 1$, then the minimum h^u for these partitions is

$$h_u(V \setminus V_2', V_2') = \max \left\{ \max_{v_i \in V \setminus V_2'} \{c(v_i)\}, \max_{v_i \in V_2'} \{c(v_i) + w(e)\} \right\} = 7 \qquad (25)$$

for V_2', while V_2'' and V_2''' have $h_u(V \setminus V_2'', V_2'') = h_u(V \setminus V_2''', V_2''') = 8$. Evidently V_2' is also a maximal partition and hence a solution for the example. The key problem, however, is to distinguish the partition V_2' already at entry $A(4, 3)$ as the number of possible partitions grows exponentially and we seek a way to compute $C(i, j)$ efficiently. At entry $A(4, 3)$ we have already three possible partitions, i.e. $V_2' \setminus v_3, V_2'' \setminus v_3$ and $V_2''' \setminus v_3$. At this point we should already be able to make a choice which partition can lead to maximal partitions at later entries such as $A(5, 4)$. Currently, we only have the maximization criteria $C(i, j)$ to evaluate partitions which does not necessarily lead to future optimal partitions as we add vertices. For the case in the example, using the maximization criteria also leads to $V_2' \setminus v_3$ being the optimal partition for $A(4, 3)$, but this need not be the case in general.

V	v_2	v_3	v_4	v_5	v_6	v_7
ρ_i	6	5	4	3	2	1
$\lambda(e_i)$	7	6	6	4	5	2
$w(e_i)$	1	1	2	1	3	1
$c(v_i)$	7	7	8	8	10	10

Table 2. Another example of vertices.

Assuming we solve the previously mentioned issue, i.e. we can compute the best partition based on the previous partitions efficiently, the results on how k_1, k_2 change upon addition or removal of vertices can be used for an attempt to find general solutions. For a set of vertices V_1 we have that $w(V_1) = w(T_1) + S(V_1)$ and hence we can chose to either keep track of the support $S_1(i, j)$ or the tail $T_1(i, j)$ as each can be computed from $A(i, j)$ and the other. For the following we choose to compute $S_1(i, j)$ explicitly. The entries for table $C(i, j)$ are also implicitly known through $A(i, j), w(V), S_1(i, j), S_2(i, j), K_1(i, j)$ and $K_2(i, j)$.

It is clearly visible in the pseudo-code from 2 at which point we assume that we have a criteria for the evaluation of partitions at $(i-1, j)$ and (i, j)

Column		1	2	3	4	5	6	7
v_i	$w(e_i)$							
v_7	1	1	1	1	1	1	1	1
v_6	3	1	1	3	4	4	4	4
v_5	1	1	2	3	4	5	5	5
v_4	2	1	2	3	4	5	6	7
v_3	1	1	2	3	4	5	6	7
v_2	1	1	2	3	4	5	6	7

Fig. 6. The dynamic programming table for the example from table 2.

Column		1	2	3	4	5	6	7
v_i	$w(e_i)$							
v_7	1	1	1	1	1	1	1	1
v_6	3	1	1	3	4	4	4	4
v_5	1	1	2	3	4	5	5	5
v_4	2	1	2	3	4	5	6	7
v_3	1	1	2	3	(4)	5	6	7
v_2	1	1	2	3	4	5	6	7

Fig. 7. Possible partitions resulting from the dynamic programming table for the example from 2. A partition is represented by a sequence of arrows where a diagonal arrow means that the vertex of the row to which the arrow is pointing is in V_2 while a horizontal arrow indicates that the vertex is in V_1.

which allows us to compute future partitions maximizing $C(i, j)$. One could use the current value of the maximization $C(i, j)$ as a heuristic. It is not clear whether $C(i, j)$ actually satisfies the aforementioned assumptions, so it is not guaranteed that a maximal partition is found. Another minor detail in the algorithm is that the assignment of the vertex in the last row v_2 may lead to K_2 increasing and needs to be dealt with separately in practical implementations. One way around this, however, is to adopt a different perspective on the problem than before. It was useful to consider vertices to be added to V_2 and removed from V_1 when searching for full and balanced partitions, but for the general case another variant seems more elegant. Instead of V we consider a subset of vertices $V^i = \{v_m, \ldots, v_{m-i+1}\}$ to be partitioned into V_1^i, V_2^i at entry $A(i, j)$. The pseudo-code from 3 shows how the formulas change when we take this approach that consider $V^i := \{v_m, \ldots, v_i\}$ to be partitioned into V_1^i, V_2^i at entry $A(i, j)$. This means when going to $A(i + 1, j)$ we either add v_i to V_1^i or to V_2^i which leads to similar formulas for both of these sets and resolves the details for assigning v_2. A solution to the problem can now be found only in the last row $m - 1$ of the dynamic programming table, namely in column j with $C(m - 1, j)$ maximal.

Algorithm 2 *Compute_table_entries_at*(i, j)

if $c_i > j$ **then**
  $[K_1(i,j), S_1(i,j)] \leftarrow [K_1(i-1,j), S_1(i-1,j)]$
  $[K_2(i,j), S_2(i,j)] \leftarrow [K_2(i-1,j), S_2(i-1,j)]$
else
  if $A(i-1, j-c_i) + c_i = A(i,j)$ **then**
    if $a - \rho_2 < K_1(i-1, j-c_i) - S(i-1, j-c_i) + c_i$ **then**
      $K_1(i,j) \leftarrow \max\{k_{q_i} | k_{q_i} < K_1(i-1, j-c_i) - S(i-1, j-c_i)\}$
      $S_1(i,j) \leftarrow w(B_{i \leq K_1(i,j)})$
    else
      $K_1(i,j) \leftarrow K_1(i-1, j-c_i)$
      $S_1(i,j) \leftarrow S_1(i-1, j-c_i) - c_i$
    end if
    if $a - \rho_i < K_2(i-1, j-c_i) - S(i-1, j-c_i)$ **then**
      $K_2(i,j) \leftarrow a - \rho_i$
      $S_2(i,j) \leftarrow c_i$
    else
      $K_2(i,j) \leftarrow K_2(i-1, j-c_i)$
      $S_2(i,j) \leftarrow S_2(i-1, j-c_i) + c_i$
    end if
    if $A(i-1,j) = A(i,j)$ and partition at $(i-1,j)$ better than at (i,j) **then**
      $[K_1(i,j), S_1(i,j)] \leftarrow [K_1(i-1,j), S_1(i-1,j)]$
      $[K_2(i,j), S_2(i,j)] \leftarrow [K_2(i-1,j), S_2(i-1,j)]$
    end if
  else
    $[K_1(i,j), S_1(i,j)] \leftarrow [K_1(i-1,j), S_1(i-1,j)]$
    $[K_2(i,j), S_2(i,j)] \leftarrow [K_2(i-1,j), S_2(i-1,j)]$
  end if
end if
return $K_2(i,j)$

5 Discussion and Conclusion

We presented a new approach for finding strategies for GRAPH-CLEAR in a tree. The new approach requires solving a partitioning problem for which we developed a formalism that leads to criteria for optimal partitions. Based on these results we presented an algorithm that computes partitions given that a full and balanced partition exist. We also presented an approach that can compute maximal partitions, but whose complexity depends on whether one can find a method to evaluate partitions at an entry (i, j) for their potential to maximize later entries in the dynamic programming table. Even without resolving this problem the algorithm returns better strategies on trees than the two previous algorithms from [8] and [7] since it degenerates to either one of the two. Given that a full and balanced partition exists we can reliably compute such a partition. It remains to test how well the general algorithm performs with heuristics, despite the fact that it will not necessarily find a

Algorithm 3 *Compute_table_entries_at*(i, j)

```
if c_i > j then
  [K_1(i,j), S_1(i,j)] ← [K_1(i−1,j), S_1(i−1,j)]
  [K_2(i,j), S_2(i,j)] ← [K_2(i−1,j), S_2(i−1,j)]
else
  if A(i−1, j−c_i) + c_i = A(i,j) then
    if a − ρ_i < K_2(i−1, j−c_i) − S(i−1, j−c_i) then
      K_2(i,j) ← a − ρ_i
      S_2(i,j) ← c_i
    else
      K_2(i,j) ← K_2(i−1, j−c_i)
      S_2(i,j) ← S_2(i−1, j−c_i) + c_i
    end if
    [K_1(i,j), S_1(i,j)] ← [K_1(i−1, j−c_i), S_1(i−1, j−c_i)]
    if A(i−1,j) = A(i,j) then
      Compare partitions when coming from A(i−1,j) with A(i−1, j−c_i)
      if A(i−1,j) leads to a better partition then
        if a − ρ_i < K_1(i−1,j) − S(i−1,j) then
          K_1(i,j) ← a − ρ_i
          S_1(i,j) ← c_i
        else
          K_1(i,j) ← K_1(i−1,j)
          S_1(i,j) ← S_1(i−1,j) + c_i
        end if
        [K_2(i,j), S_2(i,j)] ← [K_2(i−1,j), S_2(i−1,j)]
      end if
    end if
  else
    if a − ρ_i < K_1(i−1,j) − S(i−1,j) then
      K_1(i,j) ← a − ρ_i
      S_1(i,j) ← c_i
    else
      K_1(i,j) ← K_1(i−1, j−c_i)
      S_1(i,j) ← S_1(i−1, j−c_i) + c_i
    end if
    [K_2(i,j), S_2(i,j)] ← [K_2(i−1,j), S_2(i−1,j)]
  end if
end if
return K_2(i,j)
```

maximal partition when using a heuristic to choose partitions to avoid keeping track of an exponentially growing number of possible partitions. The maximization criteria may aid in this, since we are seeking final partitions which maximizes this criteria. For the first envisioned robotic applications it is estimated to have a constant bound on the number of edges per vertex, even as the number of vertices grows large. This is usually the case for two dimensional planar environments. In such scenarios the current results could already suffice for a practical implementation. Therefore, implementing and using GRAPH-CLEAR for solving target detection problems remains the main focus of further work, alongside with a continued effort to find an algorithm that computes general optimal strategies on trees.

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*, 2007. to appear.
9. 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*, 25(1):18–44, 1988.
10. L. E. Parker. Distributed algorithms for multi-robot observation of multiple moving targets. *Autonomous robots*, 12(3):231–255, 2002.
11. T. Parsons. Pursuit-evasion in a graph. In Y. Alavi and D. Lick, editors, *Theory and Applications of Graphs*. Springer-Verlag, 1976.
12. 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.

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