Autonomous Agents and Multi-Agent Systems manuscript No.
(will be inserted by the editor)

Hierarchical Visibility for Guaranteed Search in Large-Scale Outdoor Terrain

A. Kleiner · A. Kolling · M. Lewis · K. Sycara

Received: date / Accepted: date

Abstract To search for moving targets in a large area is a challenging task that is relevant in several problem domains, such as capturing an invader in a camp, guarding security facilities, and searching for victims in large-scale search and rescue scenarios. The guaranteed search problem is to coordinate the search of a team of agents in a way that all targets are guaranteed to be found. In this paper we present a self-contained solution to this problem in three-dimensional real-world domains represented by digital elevation models (DEMs). We introduce hierarchical sampling on DEMs for selecting strategical valuable locations from which larger parts of the map are visible. These locations are utilized to form a search graph from which search schedules are deduced, and agent paths that are directly executable in the terrain, are computed. Presented experimental results indicate that the proposed method leads to schedules requiring a significantly low number of agents for the search. The practical feasibility of our approach has been validated during a field experiment at the Gascola robot training site, where teams of humans equipped with IPads successfully searched for adversarial and omniscient evaders.

A. Kleiner
Computer Science Dep., Univ. of Freiburg, Georges-Koehler-Allee 52, 79110 Freiburg, Germany
E-mail: kleiner@informatik.uni-freiburg.de

A. Kolling
School of Inf. Sciences, Univ. of Pittsburgh, 135 N. Bellefield Ave., Pittsburgh, PA 15260
E-mail: andreas.kolling@gmail.com

M. Lewis
School of Inf. Sciences, Univ. of Pittsburgh, 135 N. Bellefield Ave., Pittsburgh, PA 15260
E-mail: ml@sis.pitt.edu

K. Sycara
Robotics Institute, Carnegie Mellon Univ., 500 Forbes Ave., Pittsburgh, PA 15213
E-mail: katia@cs.cmu.edu

1 Introduction

To search for moving targets in a large area is a challenging task that is relevant in several problem domains, such as capturing an invader in a camp, guarding security facilities, and searching for victims in large-scale search and rescue scenarios. Problems in which a team of searches has to be coordinated in a way that all targets are guaranteed to be found are generally referred to as guaranteed search problems. Guaranteed search makes two worst-case assumptions: first, the motion model of targets is unknown, i.e., targets are assumed to travel at infinite speeds. Second, targets are acting adversarial. Although these assumptions are very restrictive and typically used in pursuit-evasion scenarios, they are also essential in cooperative scenarios where either benevolent agents act accidentally adversarial or little is known about the targets themselves.

So far existing solutions for guaranteed search do not extend to large agent teams and besides that, are limited to two-dimensional environments [24] or certain types of idealized sensors such as unlimited range target detection [9]. Very little work has been done so far for three-dimensional search problems. A report by Lazebnik [21] discusses the challenges and complications when extending the ideas from two-dimensional visibility-based pursuit-evasion to three dimensions. In the two-dimensional case so-called critical events are defined to fully determine changes of the information about possible target locations when the searchers move through the environment. In three-dimensional environments critical events turn out to be significantly more complex since they might lead to non-local changes in the information states. As a consequence, the problem received little attention so far.

In this paper we present a novel computational solution for finding guaranteed schedules in three-dimensional real-world environments represented by elevation maps. This is carried out by extracting a search graph from the elevation map, computing the schedule requiring the least amount of agents on this graph, and then computing for each step of the schedule the time-optimal assignment of agents to vertices of the search graph. Search graphs are extracted by a hierarchical sampling method that selects from pre-classified elevation maps locations with large *detection sets*. These locations can be considered as ,,strategical valuable" since larger parts of the environment are visible from there and thus targets are more likely to be detected. We compute overlaps between these detection sets in order to determine which regions can mutually guard their boundaries. The set of strategic locations and edges extracted according to the overlaps are forming the search graph on which we compute clearing strategies for agent teams. For that purpose we introduce a variation of the Guaranteed Search with Spanning Trees (GSST) algorithm [11]. The strategy is executed by selecting for each agent at each time step an appropriate strategic location. Within the constraints of the clearing strategy we assign agents to locations in order to minimize travel time. Locations are reached by planning a trajectory executable on the terrain.

We present a self-contained solution to the problem of guaranteed search on elevation maps that finally computes for each time step of the schedule and each agent a path to execute. The use of strategic locations does not impose constraints nor does it require direct access to control inputs. This allows human searchers as well as robots to participate in the search. As shown by experimental results, this enables the direct application of the of the system in the field. The practical feasibility of our approach has been validated during a field experiment at the

Gascola robot training site, where teams of humans equipped with IPads successfully searched for adversarial and omniscient evaders. Our results indicate that the proposed method leads to schedules requiring a significantly low amount of agents for the search.

The reminder of this paper is organized as follows. In Section 2 related work is discussed. The problem is formally stated in Section 3. The approach presented in this paper is described in Sections 4, 5, and 6. Section 4 describes the generation of search graphs from elevation maps, Section 5 describes our algorithm for finding guaranteed schedules, and Section 6 describes a method to optimally assign agents at each step of the strategy to target locations. The system that has been designed for evaluating our approach in real-world environments is presented in Section 7, and results are given in Section 8. Finally, we conclude in Section 9.

2 Related Work

There is a lot of work within robotics that relates to our topic. These include visibility-based, graph-based, probabilistic and control theory approaches. Of these we will briefly review the most closely related graph-based and visibility-based work.

Most of the work on visibility-based pursuit-evasion is concerned with two-dimensional environments. In this context robots detect omniscient and fast targets with an unlimited range sensor [9]. To determine the number of robots needed for an environment is shown to be NP-hard [9]. For single robots a number of algorithms have been developed most notably one in [24] which is an on-line algorithm for a point pursuer moving in an unknown, simply-connected, piecewise-smooth planar environment. It is equipped with a sensor that only measures depth-discontinuities and can move only according to simple motion primitives (wall-following and along depth-discontinuities). Control is assumed to be imperfect control with bounded errors. Similar to prior work their approach builds a navigation graph based on the motion primitives and critical events. These events are determined from the geometry of the environment and capture how the space visible by the robot changes as it moves. Onto this graph the algorithm superimposes information states which capture whether areas in the environment are contaminated or cleared. On this graph a solution can be computed. Another approach is developed by Tovar et al. [26], who consider bounded speeds for evader and pursuer. The setting is again a simply-connected polygonal environment. The extra information about speed adds significant ,,power” to the algorithm, enabling it to compute solutions in cases where previous approaches failed. It involves the computation of a reachability set (generally an intractable problem). Modifying the evader and pursuer speed ratio relates the problem to the infinite evader speed for visibility-based problems or the 0-speed for coverage problems. One key aspect is the fact that with bounded speeds recontamination can be modeled, i.e. previously visible regions are not instantaneously recontaminated, but depending on the distance to the contaminated regions, only recontaminated after a certain time has passed. There are still quite some open questions in this direction when considering further assumptions on the evaders motion. A difficulty of the approach is to describe how the recontamination regions, so called fans, evolve with time. The main problem with visibility based approaches is that they are difficult to adapt to three dimensions or large robot teams and therefore large environments. A report by Lazebnik [21] discusses the challenges and complications when extending the ideas from two-dimensional visibility-based pursuit-evasion to three-dimensions. Especially the concept of critical events becomes difficult to tackle.

Graph-based approaches compute search strategies directly on graphs that represent an environment. This allows to utilize prior work on graph-based pursuit-evasion. Some of such problems and their applications to robotic search have been discussed in [17, 19] and [10]. Therein an emphasis is put on reducing the number of agents needed for a search strategy with all searchers moving on the graph. For search in real environments these algorithms become useful once a suitable graph can be obtained. In [20] and [16] it was demonstrated that such constructions are feasible and that we can apply graph-based search strategies to coordinate a search in real environments. These, however, only consider reducing the number of agents needed for the search and not the time this takes. Only in [3] time is considered and the authors present results on the complexity of computing strategies that minimize travel time. The travel time is given by weights on edges. It turns out that minimizing the overall travel time is already strongly NP-complete even on simple graphs such as stars and trees. One problem with modeling travel times as weights on edges, however, is that the graph on which strategies are computed usually has edges whenever contamination can spread between two vertices. Adding a travel time to such edges treats the graph like a roadmap which it may not be since it primarily captures how contamination and hence target motion can spread. Especially when dealing with complex 2.5d or 3d environments the actual best path in the map between any two vertices that are not directly connected with an edge may not correspond to a path in such a graph.

An entirely different graph-based approach is presented in [22]. Therein a graph is created by randomly sampling locations in a 2d environments and it is assumed that targets move according to a probabilistic motion model which is represented by a Markov process that determines how contamination diffuses. The graph is used for an A^* search with a suitable heuristic to search for robot paths on the graph that reduces the level of contamination. Since this approach is computationally expensive a partitioning of the environment with another heuristic is presented. The heuristic attempts to split the graph into roughly two equal parts with a minimal border. Then A^* is run on both parts sequentially while the border is guarded by some sensors. This allows five robots to search a small indoor environment.

The first paper to provide an algorithm to construct graphs for our search problem with elevation maps is [20]. Therein the graph construction is based on randomly sampling locations in the map. Every location has an associated area in which targets are detectable if an agent is placed on the location. These areas are coined *detection sets* and the detection sets of all vertices cover the entire map. Edges between vertices are created whenever two detection sets overlap in a particular manner. Since we are building directly on this work we will present it in more detail in subsequent sections.

3 Problem Description

We consider a 2.5d map represented by a height function $h : H \to \mathbb{R}^+$. The domain H is continuous and $H \subset \mathbb{R}^2$ which for all practical purpose can be approximated by a 2d grid map that contains in each discrete grid cell the corresponding height value. We write $\mathcal{E} \subset H$ for the free space in which robots can move and assume that it is connected, i.e. regardless of deployment, robots should always be able to move to another feasible point in $\mathcal{E}$. All points not in $\mathcal{E}$ are considered as non-traversable obstacles. The problem is to move a number of robots equipped with a target detection sensor through $\mathcal{E}$ to detect all targets that are potentially located therein. Targets are assumed as omniscient, and to move at unbounded speeds on continuous trajectories within $\mathcal{E}$. Additionally, targets have a minimum height h_t that can influence their visibility.

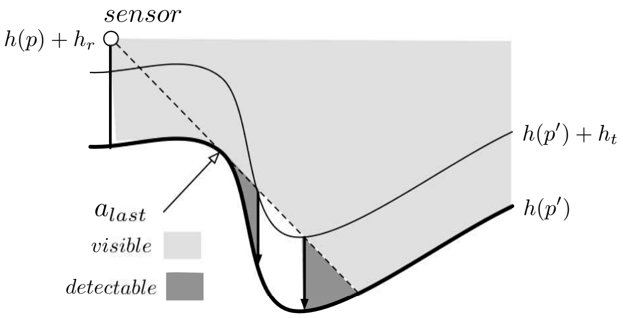


Fig. 1 An illustration how to compute detection sets for Algorithm 1.

Let $D(p) \subset H$, the detection set of a point $p \in H$, be the set of all points in H on which a target is detectable by a robot located at p. In general, $D(p)$ depends on the sensor model, height of the sensor h_r relative to $h(p)$ and height of targets h_t. We consider a limited range three-dimensional and omni-directional sensor. Hence, a target at $p' \in H$ is detectable by a robot at p if the line segment from $\{p', h(p')\}$ to $\{p', h(p') + h_t\}$ embedded in $\mathbb{R}^+$ is visible from $\{p, h(p) + h_r\}$ at distance s_r (see Figure 1 for an illustration).

Here h_t can be understood as the minimum height of any target for which we seek to guarantee a detection with the pursuit strategy. Notice that this is simply straight line visibility in 3d, which the elevation map represents. Yet, even with such a simple detection model it is not guaranteed that $D(p)$ is simply-connected nor that it is connected. This applies even if the free space of the environment in which robots and targets can move is simply-connected and also when $\mathcal{E} = H$. In this sense, our pursuit-evasion problem on elevation maps already captures significant complications that also arise in 3d pursuit-evasion.

The inclusion of target and sensor heights allows us to answer a variety of questions relating to h_r,h_t. As h_t increases the size of $D(p)$ increases as well. With $h_t = 0$ we revert back to visibility of points on the elevation map, i.e., a target is seen if the ground cell it is located on is seen. This means for practical applications that we can inform the user about the specific number of robots needed for the search given the minimal height of targets and searches.

4 Search Graph Construction

In this section we describe the process of generating $\mathcal{E}$ by classifying elevation maps into traversable terrain. We then describe our method for computing detection sets $D(p)$ for locations $p \in \mathcal{E}$. Then, we introduce two methods for generating the vertices V of search graph $\mathcal{G} = (V, E)$ from $\mathcal{E}$ by subsequently selecting locations p_i until $\mathcal{E} \setminus \bigcup_{j=1}^{i} D(p_j)$ is the empty set. These methods are a random sampling procedure and an improved method performing a depth-first search on multiple resolutions of the original map. Finally, we introduce two approaches for computing edges E of $\mathcal{G}$. The first method considers edges between any two detection sets that overlap and introduces the concept of a *shady edge*. The second method significantly reduces the number of edges and only considers overlaps between detection sets that are strictly necessary to avoid recontamination.

4.1 Height Map Classification

Free space $\mathcal{E}$, representing the area in which agents can freely move, is constructed by an elevation map classification procedue. Elevation maps are widely available on the Internet as digital elevation models (DEMs), e.g. from USGS [1], at a resolution of up to 1 meter. Higher resolutions can be achieved by traversing the terrain with a mobile robot [15]. On mobile robots elevation maps are typically computed by successively integrating three-dimensional point clouds from a tilted or rotated Laser Range Finder (LRF) with respect to the 6D pose $(x, y, d, \psi, \theta, \phi)$ of the robot.

We classify elevation maps into traversable and non-traversable terrain. The classification is carried out according to the motion model of the robot since different robot platforms have different capabilities to traverse terrain. For example, whereas wheeled platforms, such as the *Pioneer AT*, require even surfaces to navigate, tracked platforms, such as the *Telemax* robot, are capable of negotiating stairs and slopes up to 45°. Humans are capable to negotiate even steeper slopes. This specific parameters are taken into account by the classifier described in the following.

For each cell of the elevation map, representative features are created that discriminate different structure elements from the environment. We choose to use fuzzified features, which are generated by functions that project parameters, as for example, the height difference between cells, into the $[0, 1]$ interval. In contrast to binary $\{0, 1\}$ features, fuzzification facilitates the continuous projection of parameters, as well as the modeling of uncertainties. Fuzzification is carried out by combining the functions $SUp(x, a, b)$ (Equation 1) and $SDown(x, a, b)$ (Equation 2), where a and b denote the desired range of the parameter.

$$SUp(x, a, b) = \begin{cases} 0 & \text{if } x < a \\ \frac{x-a}{b-a} & \text{if } a \le x \le b \\ 1 & \text{if } x > b \end{cases} \tag{1}$$

$$SDown(x, a, b) = 1 - SUp(x, a, b) \tag{2}$$

For example, the features *Flat Surface*, and *Ramp Angle* are build from the parameters δh_i, denoting the maximum height difference around a cell, and α_i, denoting

Algorithm 1 *Detection_Set_From*(p, dir, D)

```
$\mathcal{L} \leftarrow$ set of grid cells on the line segment of length $s_r$ in direction $dir$ from $p$ ordered by distance to $p$.
$\alpha_{last} \leftarrow 0$
for $p' on \mathcal{L}$ do
    $\alpha_{tmax} \leftarrow \frac{h(p')+h_t-h(p)-h_r}{\|p-p'\|}$
    if $\alpha_{tmax} \geq \alpha_{last}$ then
        $D \leftarrow D \cup p'$
    end if
    $\alpha_{tmin} \leftarrow \frac{h(p')-h(p)-h_r}{\|p-p'\|}$
    if $\alpha_{tmin} \geq \alpha_{last}$ then
        $\alpha_{last} \leftarrow \alpha_{tmin}$
    end if
end for
```

the angle between the normal vector $\mathbf{n_i}$ and the upwards vector $(0,1,0)^T$, as shown by Equation 3 and Equation 4, respectively.

$$\delta h_i = \max_{j \text{ is neighbor to } i} |h_i - h_j| \tag{3}$$

$$\alpha_i = \arccos\left((0,1,0)^T \cdot \mathbf{n_i}\right) = \arccos\left(n_{i_y}\right) \tag{4}$$

For example, for a tracked platform we define these features by: *Flat Surface* = $SDown(\delta h_i, 0.0m, 0.8m)$, and *Ramp Angle* = $SUp(\alpha_i, 10°, 60°) \cdot SDown(\alpha_i, 25°, 40°)$. Each time the elevation map is updated, the classification procedure applies fuzzy rules on the latest height estimates in order to classify them into regions, such as *flat ground*, and *steep wall*.

Inference is carried out by the *minimum* and *maximum* operation, representing the logical *and* and *or* operators, respectively, whereas negations are implemented by $1-x$, following the definition given in the work of Elkan [8]. After applying the rule set to each parameter, the classification result is computed by defuzzification, which is carried out by choosing the rule yielding the highest output value. For discriminating more complex obstacle types, such as ramps and stairs, Markov Random Field (MRF) models, can be used [7].

4.2 Detection Set Computation

From the set of traversable cells a graph $\mathcal{G}$ has to be computed. This requires to compute a detection set for every selected location p and to remove it from $\mathcal{E}$. The detection set for location p is computed by casting rays radially from p, and to determine for each ray which points in $\mathcal{E}$ are detectable as shown by Algorithm 1. The computation is carried out by generating with the Bresenham algorithm [4] for each ray the set $\mathcal{L}$ of grid cells belonging to the line segment that starts in p with length s_r and direction dir. This set is successively traversed with increasing distance from p. For each cell $p' \in \mathcal{E}$ slopes α_{tmin} and α_{tmax} connecting p with the maximum (map elevation plus target height) of p' and minimum (map elevation) of p', are computed. Grid cells are added to detection set D as long these slopes are monotonic increasing.

Algorithm 2 *Random_Vertex_Construction()*

```
$i \leftarrow 0, V \leftarrow \emptyset$
while $E \setminus \bigcup_{j=1}^{i} D(p_j) \neq \emptyset$ do
    pick any $p_i \in E \setminus \bigcup_{j=1}^{i} D(p_j)$
    $V \leftarrow V \cup p_i,\ i \leftarrow i + 1$
end while
Return $V$
```

4.3 Random Vertex Sampling

Considering the difficulties of 3d pursuit-evasion, as well as in 2d when the environment is multiply connected, we present a first attempt to solve 2.5d pursuit-evasion by creating a graph that captures the visibility information in the environment heuristically. The graph is directly embedded into the map and each vertex is associated to a location which can be used as waypoints to plan the motion of the robot team.

We randomly select points from free space $\mathcal{E}$, i.e., the space of all traversable cells, as follows. First, pick p_1 from $\mathcal{E}$ and then subsequently pick another p_i, $i = 2, 3, \ldots$ from $\mathcal{E} \setminus \bigcup_{j=1}^{i} D(p_i)$ until $\mathcal{E} \setminus \bigcup_{j=1}^{i} D(p_i)$ is the empty set. This ensures that a target on any point in $\mathcal{E}$ can be detected from some point p_i. Finally, this procedure samples m points from $\mathcal{E}$, where each point corresponds to a vertex in set V of graph $\mathcal{G}$. Algorithm 2 sketches this procedure in pseudo code. Fig. 2 shows a few examples of such vertices and their respective detection sets. In principle, this construction does not differ significantly from basic attempts to solve an art gallery problem for complete coverage or for 2d pursuit-evasion scenarios in which graphs are constructed at random. The main difference are the detection sets $D(p)$ which we shall later use to construct edge set E to complete the graph $\mathcal{G} = (V, E)$. Notice that, due to our visibility model we have that if $h_t < h_r$, then for every pair $p, p' \in \mathcal{E}$ if $p' \in D(p)$, then $p \in D(p')$, i.e. in colloquial terms they are mutually detectable. As a simple corollary for the graph construction we get that for all i, j, $i \neq j$ we have $p_i \notin D(p_j)$.

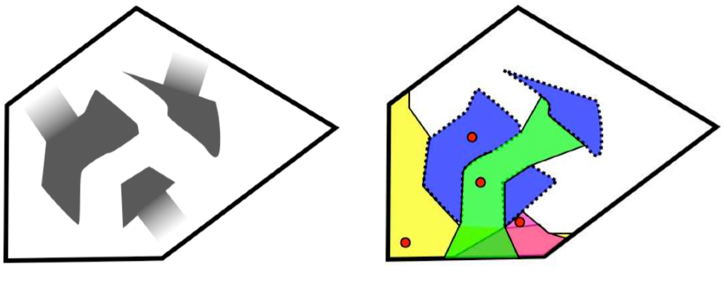

(a) (b)

Fig. 2 (a) Height map representation where grey levels correspond to height values. (b) Sampled vertices (red) with overlapping detection sets (each depicted with a different color).

4.4 Hierarchical Vertex Sampling

The main advantage of random sampling is that one does not have to compute the detection set for the majority of the points in $\mathcal{E}$, but only for those that are selected as graph nodes. Randomly selected locations, however, are not necessarily those from which larger parts of the map can be observed. They could be located in valleys or between elevated walls, thus having occluded and limited sight. A better heuristic should rather select locations with good visibility, for example, mountain peaks or bell towers located on the map. But identifying such locations with large detection sets can be computationally expensive, especially when many detection sets have to be computed in order to identify the largest ones. To tackle this problem we present a hierarchical sampling approach that automatically selects locations with large detection sets in lower resolution copies of the original map.

The hierarchical vertex sampling is carried out by generating a set of L low-resolution copies $\mathcal{M} = (M_1, ..., M_L)$ of the elevation map, where M_l denotes the map copy at level l with resolution $r_l = r_0 \frac{1}{2^l}$, and r_0 denotes the resolution of the original. For example, M_0 denotes the original map and M_L denotes the copy at the highest level. Height cells at lower resolutions are generated from higher levels by assigning the maximum of the height values from the four corresponding cells on the lower level. Figure 3 depicts the generation of two low resolution maps at level 1 and 2 from the original map.

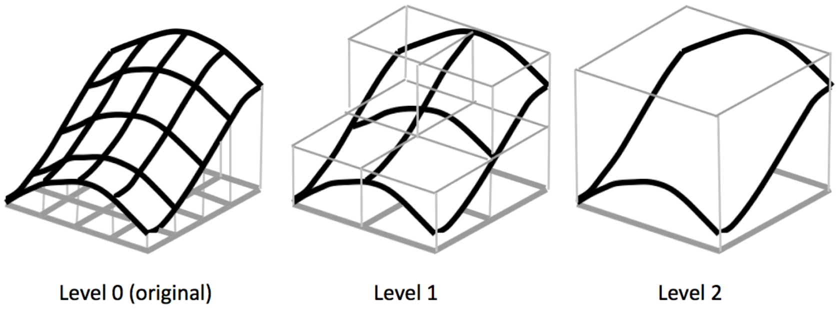


Fig. 3 Hierarchical simplification of elevation maps for computing detection sets. Level 0 represents the original map storing at each grid cell a height value. At higher levels hight values are computed by combining the height values of the four grid cells of their predecessor level by the $max()$ operator.

Likewise as shown for the random sampling procedure, the idea is to successively sample locations p_i from $\mathcal{E}$ and to remove their detection set $D(p_i)$. But instead of randomly sampling points we identify those with the largest detection set by a depth-first search on the hierarchy of $\mathcal{M}$. As shown by Algorithm 3, the search starts at the highest level L, i.e. lowest resolution of the hierarchy, by computing for each point p_L its detection[1] set $D(p_L)$. From these sets the location with the maximum detection set $p_{max} = \text{argmax}_p |D(p_L)|$ is selected. After locat-

---

[1] Note that point p_L on level L can directly be mapped to a set of points on the original map

Algorithm 3 $Hierarchical\_Vertex\_Construction()$

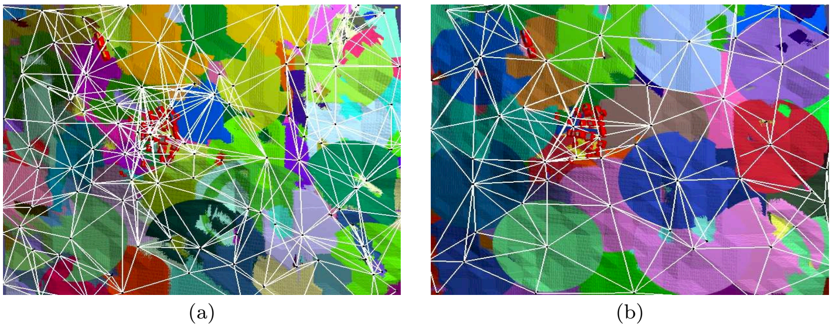
```
$i \leftarrow 0, V \leftarrow \emptyset$
while $E \setminus \bigcup_{j=1}^{i} D(p_j^L) \neq \emptyset$ do
    compute low-resolution copies $\mathcal{M} = \left(M^1, ..., M^L\right)$
    $l \leftarrow L$
    for all $p_i^l \in M_L$ do
        compute detection set $D(p_i^l)$
        $D_L \leftarrow D_L \cup \{D(p_i^l)\}$
    end for
    $p_{max} \leftarrow \text{argmax}_p \, |D_L|$
    while $l \geq 0$ do
        $p^{l-1} \Leftarrow p_{max}$ // map from $p_{max}$ to the next lower level cell
        $S_{l-1} \leftarrow \epsilon$-neighborhood $p^{l-1}$
        for all $p_i^{l-1} \in S_{l-1}$ do
            compute detection set $D(p_i^{l-1})$
            $D_{l-1} \leftarrow D_{l-1} \cup \{D(p_i^{l-1})\}$
        end for
        $p_{max} \leftarrow \text{argmax}_p \, |D_{l-1}|$
        $l \leftarrow l - 1$
    end while
    $p_i \leftarrow p_{max} \in E \setminus \bigcup_{j=1}^{i} D(p_j)$
    $V \leftarrow V \cup p_i$, $i \leftarrow i + 1$
end while
Return $V$
```

ing the maximum set on the highest level L, the search continues on lower levels in a depth-first search manner. This is carried out by computing the selection set S_{l-1} consisting of location p_{l-1} that corresponds to p_{max} from the higher level, plus further locations found around this location within a small neighborhood radius ϵ. In principle, it suffices to select ϵ in that all cells selected on $l-1$ are exactly covering p_{max} from the higher level l. However, in order to compensate for quantization errors we used $\epsilon = 4$ during our experiments. From the set S_{l-1} the best candidate of level $l-1$ is selected by computing $p_{max} = \text{argmax}_p \, |D_{l-1}|$. This procedure is continued until level 0 is reached and thus location p_{max} on the original map with maximal detection set is found. Then, $D(p_{max})$ is removed from $\mathcal{E}$ and hierarchy $\mathcal{M}$ updated accordingly. The hierarchical sampling continues until the entire map has been covered. Figure 4 depicts the result of random sampling versus hierarchical sampling on the *village map*. Hierarchical sampling leads to significantly simpler graph representations than random sampling while keeping the entire area covered.

Notice that even though we are selecting vertices with larger detection sets this is still a heuristic and by no means guarantees better strategies. Yet, we shall show in Section 8 that we indeed get a significant improvement when applying the method to diverse types of environments. Once all vertices are sampled we can proceed by adding edges between these vertices.

4.5 Shady Edge Computation

Intuitively, the edges of $\mathcal{G}$ should capture the neighborhood relationships between the detection sets $D(p_i)$. In a 2d scenario the analogue of our detection sets are



(a) (b)

Fig. 4 Example graphs on the *village* map generated (a) by random sampling, and (b) by hierarchical sampling.

guaranteed to be connected, allowing for simpler neighborhood relationships. In our case, however, these sets can be more complex. Consider the boundary of $D(p_i)$ written $\delta D(p_i)$. We are interested in vertices that can guard, i.e. avoid recontamination, of $D(p_i)$ if a robot is placed on them. Clearly, all vertices whose detection set intersects with $\delta D(p_i)$ can prevent targets from passing through aforementioned intersections. Hence, we are considering vertices v_j so that $\delta D(p_i) \cap D(p_j) \neq \emptyset$, $j \neq i$. In this case a robot on v_j can guard part of $\delta D(p_i)$. For convenience let us write $G_{i,j} := \delta D(p_i) \cap D(p_j) \neq \emptyset$ and call it the guard region of v_i from v_j. From this guard region we shall now construct two types of edges. To distinguish the types we define the following condition:

$$shady(v_i, v_j) := \begin{cases} 1 \; \exists v_{j'} \in V, j' \neq j, i : G_{i,j} \subsetneq G_{i,j'} \\ 0 \; otherwise. \end{cases} \tag{5}$$

For now suppose edges have a direction and are written $[v_i, v_j]$ from v_i to v_j. The first type, a regular edge, is created from v_i to v_j, $i \neq j$, iff $G_{i,j} \neq \emptyset$ and $shady(v_i, v_j) = 0$. In colloquial terms v_i and v_j get a regular edge if and only if there is no third vertex $v_{j'}$ whose guard region of v_i completely covers the guard region of v_i from v_j. The second type, a so called *shady* edge, is created from v_i to v_j iff $G_{i,j} \neq \emptyset$ and $shady(v_i, v_j) = 1$. In this case there is a third vertex that completely covers the guard region. Hence if $G_{i,j} \neq \emptyset$, then we have an edge $[v_i, v_j]$ that is either shady or regular. To get a graph without directional edges, written (v_i, v_j), we simply add an edge if we have either $[v_i, v_j]$ or $[v_j, v_i]$ with regular edges dominating shady edges. Write $E = E_r \cup E_s$ for the set of undirected edges where E_r are the regular and E_s are the shady edges. Algorithm 4 presents the above in details with pseudo-code. The reasoning behind creating two types of edges is straightforward. If a robot is placed at p_i, i.e. vertex v_i, it sees all targets in $D(p_i)$ and hence clears it. The robot can only be removed without causing recontamination if it can be guaranteed that no target can pass through $\delta D(p_i)$. This is satisfied if all vertices that are neighbors of v_i through regular edges are either clear or have a robot on them. The shady edges capture the remaining intersections between detection sets that are dominated by larger intersections.

Algorithm 4 *Shady_Edge_Construction*(V, P)

```
$E_r, E_s \leftarrow \emptyset, E_{r,dir}, E_{s,dir} \leftarrow \emptyset$
for $i = 1$ to $m$ do
  for $j = 1$ to $m$ do
    $I \leftarrow \delta D(p_i) \cap D(p_j)$
    if $I \neq \emptyset$ then
      if $shady(v_i, v_j)$ then
        $E_{s,dir} \leftarrow E_{s,dir} \cup \{[v_i, v_j]\}$
      else
        $E_{r,dir} \leftarrow E_{r,dir} \cup \{[v_i, v_j]\}$
      end if
    end if
  end for
end for
for $i = 1$ to $m$ do
  for $j = i + 1$ to $m$ do
    if $[v_i, v_j] \in E_{r,dir}$ OR $[v_j, v_i] \in E_{r,dir}$ then
      $E_r \leftarrow E_r \cup (v_i, v_j)$
    else if $[v_i, v_j] \in E_{s,dir}$ OR $[v_j, v_i] \in E_{s,dir}$ then
      $E_s \leftarrow E_s \cup (v_i, v_j)$
    end if
  end for
end for
Return $E_s, E_r$
```

4.6 Sparse Edge Computation

A more conservative approach is to add edges between any two overlapping detection sets only when the same part of the intersection is not covered by another detection set of a third vertex whose detection set is larger than either one of the two other vertices. This approach reduces the number of needed edges drastically. It is equivalent to creating a partition from all detection sets in which larger sets dominate smaller ones. Notice that a partition in this case is a union of non-overlapping subsets of detection sets that cover all of $\mathcal{E}$. In colloquial terms, with this approach each cell will belong to exactly one detection set and its vertex.

We will show empirically in Section 8 that this reduction has a positive impact on the strategy computation even with the given constraint of contiguous strategies.

5 Strategy Computation

The search graph $\mathcal{G}$ constructed in Section 4 represents strategic locations in $\mathcal{E}$ as vertices and their neighborhood relations as edges. The goal of this section is to describe an algorithm that coordinates the movements of all agents in order to clear $\mathcal{E}$. For this purpose we denote vertices occupied by agents as *guarded*, and define contamination on $\mathcal{G}$. The relation between $\mathcal{G}$ and $\mathcal{E}$ is straightforward: Placing agents on vertices v_i in $\mathcal{G}$ corresponds to agent movements towards associated waypoint locations p_i in $\mathcal{E}$.

Definition 1 (Vertex Contamination) A vertex $v \in \mathcal{G}$ is cleared if it is guarded. It is recontaminated if it is not guarded and there exists a path on $\mathcal{G}$ consisting of regular edges and unguarded vertices between v and a contaminated vertex v'. If all $v \in \mathcal{G}$ are cleared then $\mathcal{G}$ is cleared.

Such a contamination definition is common in graph-searching, with the exception that it only spreads via regular edges and that we only consider vertices. Another important difference to previous graph-searching problem is how we will define strategies, i.e. the sequences of moves that clear $\mathcal{G}$.

Definition 2 (Strategy) A strategy S consist of n_s steps. Step i starts at time $t_i \in \mathbb{R}$ and $t_i < t_{i+1}$. Each step consist of the following moves:

1. At time t_i available agents that are not guarding vertices can be placed onto new vertices.
2. At time $t'_i,\ t_i < t'_i < t_{i+1}$, agents guarding vertices can be removed.
3. At time $t''_i,\ t'_i < t''_i < t_{i+1}$, contamination spreads.

A strategy that clears an initially fully contaminated $\mathcal{G}$ with the minimum number of agents guarding at any time t is a minimal strategy.

This definition reflects the fact that an agent removed from a vertex cannot be reused immediately since it has to move through $\mathcal{E}$ before it can guard another vertex. Furthermore, we do not allow so called *sliding* moves which are common in graph-searching. In our context such a move would allow an agent to guard a vertex and then slide along an edge to a neighboring vertex, guarding and clearing it. The problem is that such a move is not atomic[2] in $\mathcal{E}$. It takes time for an agent to travel between vertices and during this time we cannot guarantee that the contamination from the new vertex spreads to the previously guarded vertex. This is hard to guarantee even in a 2d scenario but almost impossible in 2.5d.

In order for the above to be useful for clearing $\mathcal{E}$ we now address the relationship between clearing $\mathcal{G}$ and $\mathcal{E}$. Contamination on the graph is more conservative than in $\mathcal{E}$, i.e. we are going to show that if we compute a strategy with k agents that clears the graph then we can clear $\mathcal{E}$ also with k agents. A strategy $S_{\mathcal{E}}$ in $\mathcal{E}$ is defined identical to those on graphs but with vertices replaced by their locations. Hence placing an agent on $v \in \mathcal{G}$ is to place an agent onto its associated position $p \in \mathcal{E}$ which clears $D(p) \subset E$. Likewise executing a strategy S on $\mathcal{G}$ represents executing a strategy $S_{\mathcal{E}}$ in $\mathcal{E}$ by visiting all associated locations.

To make the following results consistent with the sparse edge computation we introduce the *associated* detection set $\bar{D}(p) \subseteq D(p)$ for a vertex v. In colloquial terms, $\bar{D}(p)$ is the area in $\mathcal{E}$ that the agent on vertex v is responsible for. For the sparse edge computation, suppose we have $v_1, \ldots, v_n$ ranked by the size of $D(p_i)$ in decreasing order. Now, $\bar{D}(p_i) := D(p_i) \setminus \bigcup_{j=1}^{i-1} D(p_j)$. If we are not using sparse edge computation, then $\bar{D}(p_i) := D(p_i)$, i.e. an agent is fully responsible for the entire detection set.

Lemma 1 *If during the execution of a strategy S we have $v \in \mathcal{G}$ cleared then $\bar{D}(p) \subset \mathcal{E}$ is cleared for $S_{\mathcal{E}}$.*

Proof: Clearly, when an agent is placed on v to clear it in a step of a strategy $\bar{D}(p)$ is also cleared. Hence, we have to show that if v remains cleared at subsequent steps then so does $\bar{D}(p)$. We shall achieve this with an inductive argument across steps of the strategy.

---

[2] In the sense that multiple recontamination events in $\mathcal{E}$ can occur in the meantime.

Let v be a vertex whose agent gets removed at step s and let s be the first step that removes an agent. Suppose (by assumption of the lemma) v does not get recontaminated at step s at time t''_s.

Consider $\delta\bar{D}(p)$ where δ denotes the boundary of a set in $\mathcal{E}$. If all regular neighbors of v are guarded, then $\delta\bar{D}(p)$ is detectable since $\delta\bar{D}(p) \subset \bigcup_{v' \in E_r(v)} D(p')$ and hence $\bar{D}(p)$ remains clear. We have

$$\delta\bar{D}(p) \subset \bigcup_{v' \in E_r(v)} D(p') \tag{6}$$

by construction of regular edges[3].

This simple argument can also be applied to a set of vertices as follows. Let $N_{unguarded} \subset V$ be all unguarded neighbors reachable via regular and unguarded paths from v. By definition of recontamination if any of these neighbors is contaminated then so is v. Hence all vertices in $N_{unguarded}$ are cleared. Furthermore, all regular neighbors of $N_{unguarded}$ in $V \setminus N_{unguarded}$ are guarded. Let $N_{guarded}$ be all guarded vertices at step s at time t''_s. Therefore, $\delta\left(\bigcup_{v' \in N_{unguarded}} \bar{D}(v')\right) \subset \bigcup_{v' \in V_{guarded}} D(v')$ and hence no contamination in $\mathcal{E}$ can enter $\bigcup_{v' \in N_{unguarded}} \bar{D}(v')$. Hence if v is clear at step s at time t''_s then so is $\bar{D}(p)$.

Continuing this argument by induction for subsequent steps proves the claim since for every subsequent step $s+1$ we can assume that if v clear then $\bar{D}(p)$ clear for all v from $N_{unguarded}$ from step s. $\square$

Theorem 1 *If S clears $\mathcal{G}$ then $S_{\mathcal{E}}$ clears $\mathcal{E}$.*

Proof: The theorem follows directly from the lemma and the fact that $\mathcal{E} \subseteq \bigcup_{i=1}^{n} \bar{D}(p_i)$, i.e. if all v_i are clear, all $\bar{D}(p_i)$ are clear and hence $\mathcal{E}$ is clear. $\square$

So if we compute a strategy S for $\mathcal{G}$ and execute its corresponding $S_{\mathcal{E}}$ in $\mathcal{E}$ we clear $\mathcal{E}$ and detect all targets therein. In the following section we address the problem of computing strategies for our graph version of the problem.

5.1 Algorithm

Our problem on $\mathcal{G}$ is very similar to the prior edge-searching problem as defined by Parson [23]. In fact, we can adapt an algorithm from [2] that computes contiguous strategies without recontamination for edge-searching on trees, i.e. a strategy that guarantees that all cleared vertices are a connected subtree. Such strategies have the practical advantage that the cleared area is relatively compact, although it may not necessarily be connected in $\mathcal{E}$. In contrast, non-contiguous strategies allow placement of agents far from the currently cleared area and hence can lead to long paths through contaminated areas. The algorithm was originally developed to handle a case in which multiple agents are required to clear a single vertex. But turns out not to be optimal for this purpose [18]. Yet, for the simpler unweighted case the resulting contiguous strategies are in fact optimal on trees. In [14] it was shown how to use a labeling-algorithm similar to [2] and adapt it to strategies on graphs by trying many spanning trees. This procedure can be asymptotically optimal for the graph given that enough spanning trees and strategies on each spanning tree are tried.

---

[3] This is straightforward to see by supposing the contrary, i.e. $\delta\bar{D}(p) \setminus \bigcup_{v' \in E_r(v)} D(p') \neq \emptyset$ in which case the point $x \in \delta\bar{D}(p) \setminus \bigcup_{v' \in E_r(v)} D(p')$ has to lead to a regular edge by definition

The key differences between edge-searching and our variant is that we disallow sliding moves, apply our contamination between removal and placement of agents and are only concerned with contamination on vertices. So let us assume that we converted $\mathcal{G}$ into a tree by selecting a spanning tree T. For now let us also ignore the difference between shady and regular edges and defer its discussion to Section 8. The following describes the adaptation of the label-based algorithm from [2].

We define a directional label for every edge $e = (v_x, v_y)$. For the direction from v_x to v_y we write $\lambda_{v_x}(e)$. This label represents the number of agents needed to clear the subtree rooted at v_y and created by removing e. It is computed as follows: If v_y is a leaf then $\lambda_{v_x}(e) = 1$. Otherwise let $v_1, \ldots, v_m$ be the $m = degree(v_x) - 1$ neighbors of v_y different from v_x. Define $\rho_i := \lambda_{v_y}(\ (v_y, v_i)\ )$ and order all $v_1, \ldots, v_m$ with ρ_i descending, i.e. $\rho_i \geq \rho_{i+1}$. The team of robots now clears the subtrees that are found at each v_i in the order $v_m, \ldots, v_1$. Notice that this is the optimal ordering given that the strategy has to be contiguous and without recontamination. This leads to an overall cost that we associated to $\lambda_{v_x}(e)$. In original edge searching in [2] we would have $\lambda_{v_x}(e) = \max\{\rho_1, \rho_2 + 1\}$. In our modified version this equation becomes:

$$\lambda_{v_x}(e) = \begin{cases} \rho_1 + 1 & if\ \rho_1 = 1 \\ \max\{\rho_1, \rho_2 + 1\} & otherwise \end{cases} \tag{7}$$

Where we assume that $\rho_2 = 0$ if $m = 1$. The change results from the fact that the guard on v_y can be removed only after the first vertex of the last subtree, i.e. v_1, is cleared. This is only a concerne when $\rho_1 = 1$, i.e. v_1 is a leaf. Otherwise, if $\rho_1 > 1$, the guard can be removed right after v_1 is guarded and used subsequently in the remaining subtree beyond v_1, not leading to a higher cost than in edge-searching. For edge-searching the guard on v_y can instead be moved into v_1 via a sliding move to clear it which leads to lower cost for clearing leaves. From this it follows that on the same tree the edge-searching strategies and our modified variant can only differ by one agent.

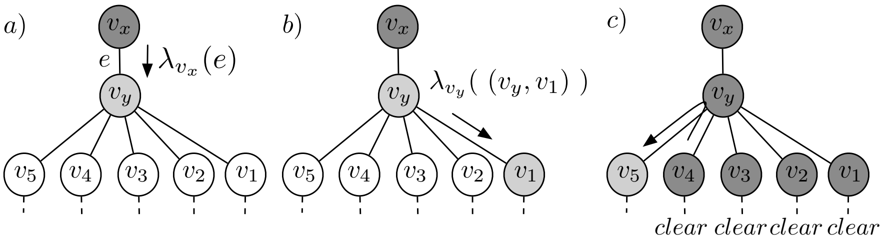

a)
v_x
e
$\lambda_{v_x}(e)$
v_y
v_5 v_4 v_3 v_2 v_1
b)
v_x
v_y
$\lambda_{v_y}(\ (v_y, v_1)\ )$
v_5 v_4 v_3 v_2 v_1
c)
v_x
v_y
v_5 v_4 v_3 v_2 v_1
clear clear clear clear

Fig. 5 An illustration of the label computation. Part a) shows an agent placed at v_y followed by part b) in which the team starts clearing the subtrees until in part c) they enter the last subtree. White vertices are contaminated, light grey vertices are guarded, and dark grey vertices are cleared.

Algorithm 5 *Compute_Strategy*$(G, trees)$

```
$max\_cost \leftarrow \infty$
for $i = 1$ to $trees$ do
    Generate a spanning tree $T$ from $G$
    Compute strategy $S_T$ on $T$
    Convert $S_T$ to a strategy $S_G$ on $G$
    if $cost(S_G) < max\_cost$ then
        $best\_strategy \leftarrow S_T, max\_cost \leftarrow cost(S_G)$
    end if
end for
Return $best\_strategy$
```

Once all labels are computed we can determine the best starting vertex and from there a sequence in which all vertices have to be cleared. This can be done in a straightforward manner by simply following clearing subtrees recursively and ordered by ρ as described above. The result is a sequence of vertices that represents the strategy. Notice that for any strategy that places multiple agents at one time step we can find an equivalent strategy (i.e. one that clears vertices in the same order with the same number of agents) that places exactly one agent per time step. Hence it suffices to consider strategies that place only one agent per step.

Our formulation allows us to use the idea from the anytime algorithm, called GSST, from [10] which tries multiple spanning trees T to improve the strategy for the graph. For this we generate a number of spanning trees for our graph G and compute a strategy for each. These we convert to strategies on the graph by leaving agents at their position whenever a cycle edge leads to a contaminated vertex. Finally, we select the graph strategy across all spanning trees that leads to the least robots that are needed on the graph. Algorithm 5 sketches this idea in pseudo code. Results presented in Section 8 confirm that this method works well in practice and with graphs constructed from real environments.

6 Task Assignment

Given a pursuit-evasion strategy that requires k agents, written $a_1, \ldots, a_k$, we will now compute an assignment of the guarding tasks to agents and attempt to minimize the time it takes for all agents to execute the strategy. In our case a contiguous strategy is given by a sequence of vertices that need to be guarded. Let us write $v_1, \ldots, v_n$ for this sequence. Once a vertex has no contaminated neighbors anymore its guarding agent is free to move to another vertex without incurring recontamination. This occurs precisely when the last neighboring vertex is guarded and thereby cleared from contamination. We can hence generate a task τ_i for every $i = 1, \ldots, n$ that starts at step i and terminates after some step $j \geq i$, i.e. the agent is released at step j when task τ_j is started. In principle this conversion can be applied to other types of pursuit-evasion strategies such as Graph-Clear [19] which involves actions other than guarding as well as actions on edges.

We shall now define a task $\tau_i := (l_i, d_i)$ as a tuple of a location l_i that corresponds to the location of vertex v_i on map H and d_i which is the step until l_i needs to be occupied. Here the cost for executing τ_i, and thus the time needed for reaching vertex location l_i, is computed by A* planning on the elevation map with respect to the current location of the assigned agent. The sequence of tasks is entirely determined by our strategy, but the assignment of agents to these tasks is not.[4]

To complete a task $\tau = (l, d)$ an agent a needs to arrive at location l and occupy it until we reach step d. Step d is completed once all locations $l_j, j \leq d$ have been reached (although some of the agents may already be released from these locations). Once step d is completed, agent a can continue moving towards another task location. Some task locations are hence be occupied in parallel since multiple agents may be waiting for their release. By construction the total number of agents occupying task locations will not exceed k. Figure 6 illustrates the new task sequence arising from a strategy.

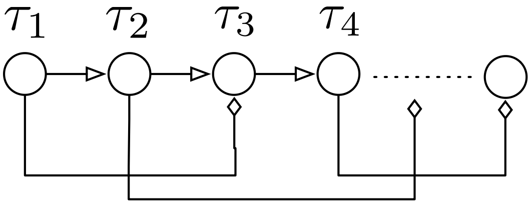


Fig. 6 A sequence $\tau_1, \tau_2, \tau_3, \ldots$ of tasks arising from a strategy, where circles denote tasks and edges towards a diamond denote task dependencies (i.e. mutual guarding constraints) which determine when agents on the associated locations can be removed. For example, at step 3 location l_1 from task $\tau_1 = \{l_1, d_1 = 3\}$ can be released because then location l_3 of task τ_3 has been reached by an agent. Hence, the agent from l_1 may be used for τ_4 during next step 4.

The overall execution time for the strategy is determined by the speed at which agents can travel to the locations of their assigned tasks with each agent's time at the location depending on other agents. Let us now briefly formalize the problem.

Definition 3 (Task Assignment) A task assignment is a surjective function $\mathcal{A} : \{\tau_1, \ldots, \tau_n\} \to \{a_1, \ldots, a_k\}$ with the following property: if $\mathcal{A}(\tau_i) = \mathcal{A}(\tau_j)$ for some $j > i$ then $d_i < j$.

In colloquial terms, this definition just ensures that every agent has at least one task and that an agent cannot be assigned to another task before it is released. To formalize the contribution of travel time let us write $a(t)$ for the location of agent a at time t. Further, write $\mathcal{T} : \mathcal{E} \times \mathcal{E} \to \mathbb{R}^+$ to represent a path planner (in our experiments in Section 8 this will be an A* planner) that returns the time it takes for an agent to travel between two locations in $\mathcal{E}$ written $\mathcal{T}(l, l')$. Write t_i be the time at which step i is completed. We can now define t_i inductively via $t_0 := 0$ and

$$t_{i+1} := t_i + \mathcal{T}(\mathcal{A}(\tau_{i+1})(t_i), l_{i+1}). \tag{8}$$

Notice that the term $\mathcal{T}(\mathcal{A}(\tau_{i+1})(t_i), l_{i+1})$ may well be 0 if the agent $\mathcal{A}(\tau_{i+1})(t_i)$ is already on l_{i+1} at time t_i. In fact, with a larger number of agents we should expect this to occur frequently as agents are moving in $\mathcal{E}$ simultaneously. Figure 7 shows three steps that finish at the same time, i.e. $t_3 = t_4 = t_5$.

---

[4] Note that there are pursuit-evasion problems and algorithms that immediately assign an agent to an action, but to our knowledge there are none that consider the number of agents as well as execution time with an underlying path planner.

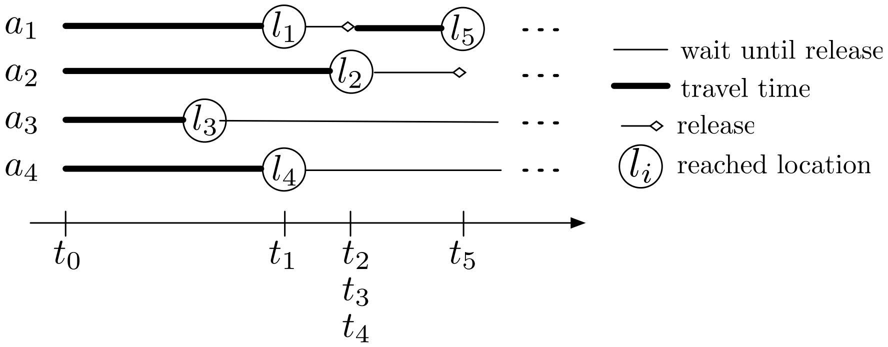


Fig. 7 Agents a_1, a_2, a_3 and a_4 move to locations l_1, l_2, l_3 and l_4 respectively. Step 1 is completed once a_1 reaches l_1. Other agents may already be at their assigned locations at this time. At step 2 agent a_1 is released and proceeds to l_5. Since a_3 and a_4 have already reached their task locations at t_2 we have $t_2 = t_3 = t_4$. Agent a_2 is released once a_1 reaches l_5 and so on.

Obviously, the above assumes that agents actually move towards their next assigned tasks immediately after release. For an agent a let $\mathcal{A}|_a := \{\tau \mid \mathcal{A}(\tau) = a\}$ be the set of all tasks assigned to a, ordered with their index ascending as before. For convenience let us write $\mathcal{A}|_a = \{\tau_1^a, \tau_2^a, \ldots, \tau_{n_a}^a\}$ where $n_a = |\mathcal{A}|_a|$. At time t_0 every agent a immediately moves towards their first task $\tau_1^a = (l_1^a, d_1^a)$ following the planner and needing $\mathcal{T}(a(t_0), l_1^a)$ time units and at every subsequent release they move immediately towards the next assigned task location. We can now formalize our main problem:

Definition 4 (Minimal Assignment) Given a fixed $\mathcal{T}$ a sequence of tasks $\tau_1, \ldots, \tau_n$ and agents $a_1, \ldots, a_k$ let the minimal assignment $\mathcal{A}_{min}$ be such that:

$$\mathcal{A}_{min} = argmin_{\mathcal{A}}\{t_n\} \tag{9}$$

To compute a task assignment $\mathcal{A}$ it is helpful to compress the notation. Instead of the sequence of tasks we now consider sets of tasks whose locations have to be reached before another task is released. This is useful because between releases we have a constant number of agents available and a constant number of tasks that all have to reached before new agents become available.

Let us write $\tilde{t}_0, \tilde{t}_1, \ldots, \tilde{t}_{\tilde{n}}$, for the times at which at least one agent is released. At $\tilde{t}_0$ all agents are free and can be assigned to tasks. Write $\tilde{F}_0 = \{1, \ldots, k\}, \ldots, \tilde{F}_{\tilde{n}}$ for the set of free agents after the step completing at $\tilde{t}_i$. Let $\tilde{T}_i$ be all tasks that have an agent on their location at time $\tilde{t}_i$, $i = 1, \ldots, \tilde{n}$.

This compressed notation gives us an immediate first insight. Namely, to minimize the time difference $\tilde{t}_i - \tilde{t}_{i-1}$ we have to solve a Linear Bottleneck Assignment Problem (LBAP) and match some agents from F_{i-1} to the new tasks $\tilde{T}_i \setminus \tilde{T}_{i-1}$, $i = 1, \ldots, \tilde{n}$. The cost of an assignment between a free agent a and a task $\tau = (l, d)$ is simply given by the difference between $\tilde{t}_{i-1} - t_{arrival}$ where $t_{arrival}$ is the earliest time, as determined by $\mathcal{T}$, at which a can be at l, i.e. $t_{arrival} = t_{last} + \mathcal{T}(a(t_{last}), l)$ where t_{last} is the time at which a became released and hence free. Using this we can build a cost matrix $c(a, \tau)$ to capture the cost of each possible assignment. Note that $|F_{i-1}| \geq |\tilde{T}_{i+1}|$ and we have to add an idle task τ_0 so that the LBAP would assign some robots to a dummy task τ_0 that the agent simply ignores when moving to the next location. From here on any LBAP algorithm can be applied to minimize $\tilde{t}_i - \tilde{t}_{i-1}$ and for $\tilde{t}_i$ this would give us the minimum possible value, given that $\tilde{t}_{i-1}$ was fixed. But this does not guarantee that $t_n = \tilde{t}_n$ is minimal and brings us directly to the main problem which is best illustrated with the following example.

Suppose we have four agents a_1, a_2, a_3 and a_4 and $\tilde{T}_1\{\tau_1, \tau_2\}$ with $\tau_1 = \{l_1, 2\}, \tau_2 = \{l_2, 4\}$ and $\tilde{T}_2 = \{\tau_3, \tau_4\}$ with $\tau_3 = \{l_3, 4\}, \tau_4 = \{l_4, 4\}$. Fig 8 shows the locations of the agents and $l_1, \ldots, l_4$ and two different assignments for $\tilde{F}_0$ on $\tilde{T}_1$ that in turn allow different assignment for F_1 onto $\tilde{T}_2$. The assignments are also shown in Figure 9. It is easy to see that an optimal solution to the LBAP for $\tilde{F}_0$ on $\tilde{T}_1$ leads to an overall worse solution. In colloquial terms, we can sacrifice some time in an assignment at one step and instead choosing to give an idle task to an agent that will travel to its tasks for a subsequent assignment and thereby improving it. This can lead to overall less time spent, i.e. a smaller t_n.

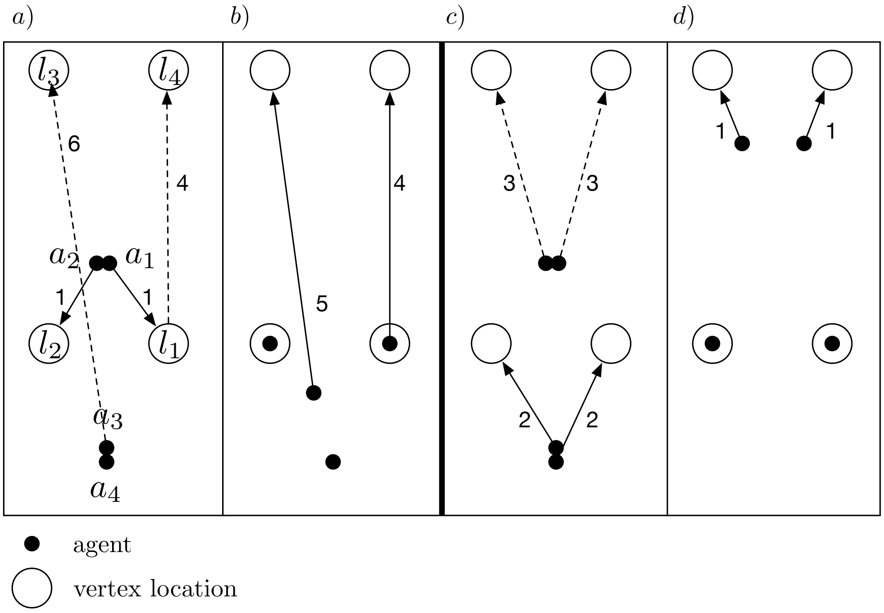


Fig. 8 Each part a)-d) shows four locations and agents. Part a) and b) show one assignment in which agents a_1 and a_2 move to l_1 and l_2, the optimal assignment to minimize $\tilde{t}_1$. After $\tilde{t}_1$ agent a_2 is released and at this point assigning a_2 and a_3 is the optimal assignment to minimize $\tilde{t}_2$ given that $\tilde{t}_1$ is fixed. Part c) and d), however, show an assignment that leads to a larger $\tilde{t}_1$ but smaller $\tilde{t}_2$.

The dependency between subsequent assignments is due to the fact that some robots can be assigned to tasks for future steps if previous steps do not utilize all agents. If at every step the number of tasks is equal to the number of agents, then the repeated solving of the linear bottleneck assignment problem (LBAP) will yield an optimal solution. Otherwise, from a global perspective, the repeated computation of locally optimal LBAP solutions is a greedy algorithm.

Figure 9 shows the assignment of agents to tasks in a familiar manner for LBAP problems in the form of consecutive bipartite graphs. Repeated assignment problems are also known as multi-level assignment problems and one variant that has some resemblance to our problem is presented in [6]. Unfortunately it is $NP-$*complete* and we conjecture that this may be the case for our minimal assignment as well. A detailed exposition is, however, beyond the scope of this paper and for our purposes the presented approach to solve multiple LBAPs is sufficient and in Section 8 we shall see that this already leads to a significant improvement. Note that for practical purposes agents can also be assigned at each step of the compressed schedule in polynomial time by the method presented in [13]. Other methods are presented in the survey [5]. However, this assignment is suboptimal in terms of execution time since it ignores dependencies of subsequent steps in the schedule.

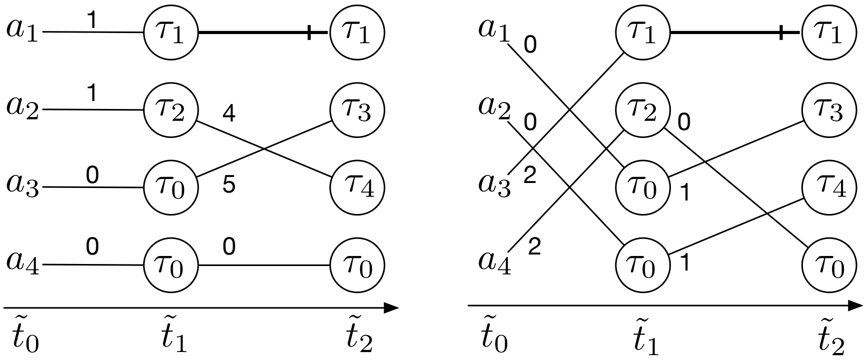
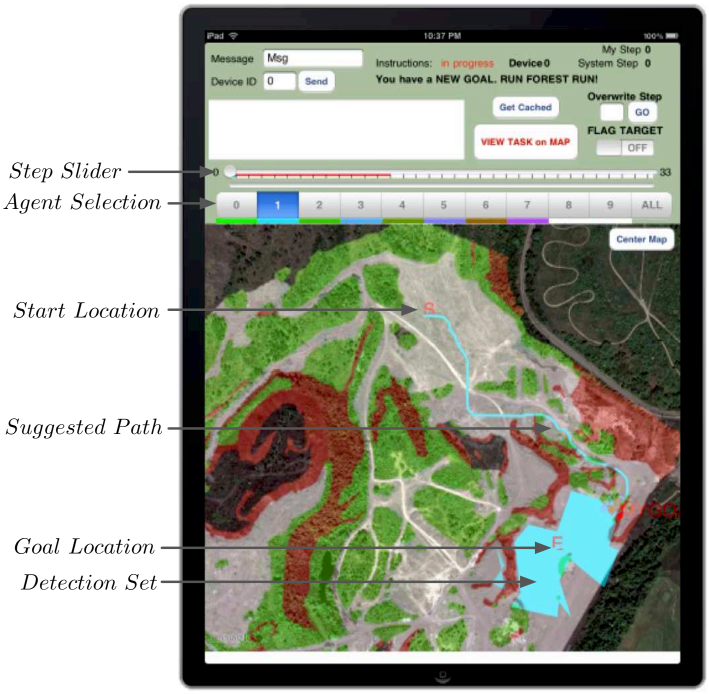


Fig. 9 Two task assignments visualized as graphs that correspond to Figure 8. The agent assigned to τ_1 has to remain there until release while τ_2 can be reassigned in the second level LBAP. The consecutive LBAP solution for both levels shown on the left is, however, not optimal for $\tilde{t}_2$.

7 Real-World Interface

Few of the prior work on searching for moving targets or pursuit-evasion has ever been tested in real world applications, especially not in large and realistic environments. One main obstacle is the integration of all aspects of the problem from mapping up to the computation and coordinated execution of search strategies. In this section we describe a system that integrates all solutions to these problems presented here and in previous work and guides searchers through a large outdoor environment.

Using an annotated elevation map we constructed a graph representation as described in Section 4 but taking into account additional obstructions for visibility due to cluttered terrain. On this graph we computed a strategy, using the algorithm presented in Section 5 and a corresponding task assignment and schedule following the procedure described in Section 6. This schedule is then made available to all agents. For our demonstration we used human agents equipped with mobile devices (IPads) on which we programmed a custom Objective-C application. All devices were communicating via a 3G connection and all data was logged at a central server. The interface of the application is shown in Figure 10. All searching agents had information about the instructions of all other searching agents and their locations by exchanging GPS data at two second intervals. The evading agents, i.e. targets, were also given a device each to simulate omniscient evaders. They were omniscient because they were able to see on the device in real-time locations of all the searchers but also other evading agents. When a searching agents saw an evader they logged the encounter by tagging the respective area on the map via the interface. All agents received a warning signal if their GPS indicated that they were close to terrain that was classified as non-traversable. Once agents reach their assigned locations for a execution step, the system sends messages to other agents informing them about the progression and assigns them subsequently to their new tasks. Figure 11 gives an overview on the main parts of the system.



Message Msg

Device ID 0

Instructions: in progress Device 0

My Step 0

System Step 0

You have a NEW GOAL. RUN FOREST RUN!

Step Slider
Agent Selection
Start Location
Suggested Path
Goal Location
Detection Set
iPad
10:37 PM
100%
Send
Get Cached
VIEW TASK on MAP
Overwrite Step
GO
FLAG TARGET
OFF
0
33
0
1
2
3
4
5
6
7
8
9
ALL
Center Map
S
E

Fig. 10 The IPad interface for all agents. Additionally each agent receives real-time locations and plans of other searchers that are directly displayed on the device. Evading agents receive the same information but additional also information about other evaders.

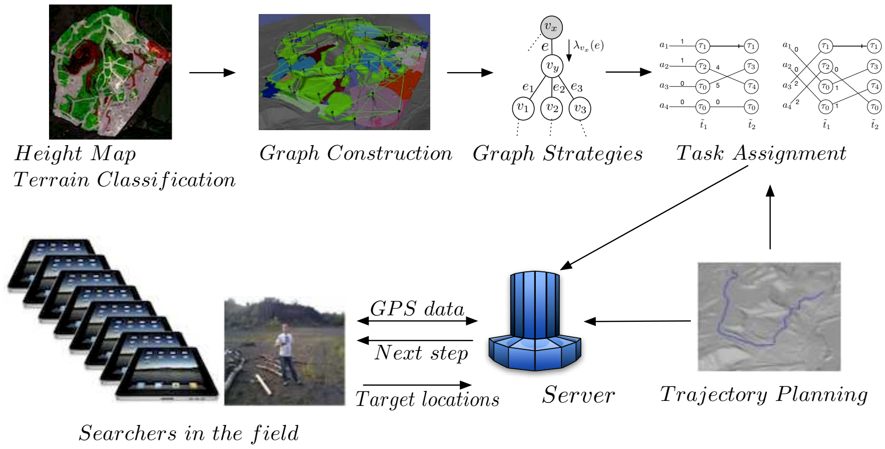

Height Map
Terrain Classification
Graph Construction
Graph Strategies
v_x
e
$\lambda_{v_x}(e)$
v_y
e_1
e_2
e_3
v_1
v_2
v_3
Task Assignment
a_1
a_2
a_3
a_4
τ_1
τ_2
τ_3
τ_4
τ_0
i_1
i_2
GPS data
Next step
Target locations
Server
Trajectory Planning
Searchers in the field

Fig. 11 Overview on the main parts of the system: A search graph is constructed from a classified elevation map, from which a graph strategy and task assignment are computed. This data is then used by searchers and evaders equipped with IPads that are connected with a central server.

8 Experimental Results

In this section we present results from applying and evaluating our graph construction approach for the computation of pursuit-evasion strategies on elevation maps. First, in Section 8.1 we are investigating the performance of the graph construction based on random sampling on three sample maps. We further investigate the question of whether shady edges should be considered when computing contiguous strategies in order to yield fewer strategies with fewer robots. Second, in Section 8.2 the impact of hierarchical sampling and sparse edges is examined on large-scale maps. Finally, in Section 8.3 results from deploying the system in the field, the *Gascola* outdoor area around Pittsburgh, are presented.

8.1 Random Sampling with Shady Edges

In this section we will clarify the role of shady edges for the computation of strategies on graphs and address issues first raised in [20].

Recall the definition of regular edges, E_r, and shady edges, E_s, from Section 4.5. Both types of edges capture some aspects of the neighborhood relations between detection sets. From our formulation of the graph model in Section 5 and Eq. 6 from Lemma 1 we know, however, that regular edges suffice to guard the boundary of an associated detection set. Hence, shady edges only capture visual proximity but are not directly relevant for spreading contamination in $\mathcal{E}$. But visual proximity can be important for contiguous strategies. The advantage of contiguous strategies is that the set of cleared vertices is connected which translates to a rather compact cleared area in $\mathcal{E}$. By considering shady edges the number of possible contiguous strategies can increase and possibly include one that is better than contiguous strategies based only on regular edges. For non-contiguous strategies, it is obvious that we only have to consider regular edges since agents are not constraint to guard only vertices that are adjacent to already guarded vertices. In this case, edges only matter for contamination and not to constrain movement so that the cleared are is more compact. The question how to treat shady edges was first raised in [20] and investigated experimentally. Combining our work from Section 5 with Algorithm 5 we now resolve the issue more precisely.

Consider Algorithm 5 from Section 5. Recall that it computes strategies on a generated spanning tree T and then converts these to graph strategies by having agents guard a vertex until all its neighbors in G are cleared. From this it follows that the contiguity requirement can only have an impact on T since the strategy is already required to be contiguous on T (which translates to contiguity in G). Hence shady edges can only have a possibly positive impact when included in T. During the conversion of the strategy from T to G they should not be considered since they can only worsen the strategy on G by requiring agents to guard vertices longer. So we can answer the question whether shady edges should be considered during the conversion to the negative and thereby superseding the experimental verification from [20] which showed that treating regular and shady edges equally leads to worse strategies. The key observation is that shady edges are not required in Eq. 6.

The consideration of shady edges for constructing T, however, needs to be verified experimentally. Excluding them from T would additionally constrain the motion of robots between vertices whose detection sets overlap and are hence within visual proximity. This leads us to define the following two variants for the strategy computation. For variant *sdy* we generate random depth-first spanning trees considering all edges from E. For variant *reg* we only consider regular edges for the spanning tree. %todoHere I am a bit confuses:) sdy should mean to produce shady edges and to ignore them during the strategy computation. For both variants we compute strategies on the spanning tree T as presented in Section 5 and convert them to strategies on G as follows. Robots continue guarding a vertex not only until all neighbors in T are cleared but until all neighbors considering edges from E_r are cleared. Hence, shady edges have no significance for the conversion and only in variant *sdy* they can be part of T.

Some of the experimental results from prior work in [20] did in fact address the question whether *sdy* or *reg* leads to generally better strategies. In [20] this was denoted as a *bias* in the spanning tree generation and in what follows we will adapt the relevant results from [20] to our context. The experiments were carried out by randomly sampling vertices on three maps depicted in Figure 12. The resolution of (a) and (b) is 0.1 units/pixel, and 10 units/pixel for (c). Sensing ranges mentioned below are always measured in units. The elevation of each cell in the map is given by its grey level and ranges from 0 to 10 units with 0 represented as white and 10 as black. Traversability classification is always based on the model of an *all-terrain* robot. Due to the random components of the algorithm, which are the random sampling of the graph structure and the strategy computation based on choosing from multiple random spanning trees, all presented results are averaged across 100 randomly generated graphs.

Further experiments from [20], carried out with the now superseded variant that considers shady edges equal to regular edges, revealed the effects of modifying the number of spanning trees, sensing range and target and sensor heights. A first set of tests was conducted on the *Sample map* from Figure 12(a). For each randomly sampled graph the best strategy was computed based on 100, 1000,

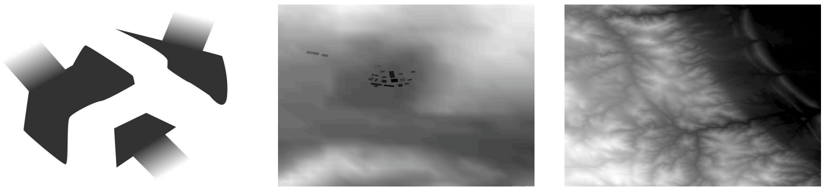


Fig. 12 Height maps for testing: (a) *Sample map* with a three-way canyon, three plateaus with each having its own ramp and several concave sections (843x768 cells). (b) Map of a small village with surrounding hills (798x824 cells). (c) Map of a mountain area located in Colorado, US (543x699 cells).

trees	variant	min	max	mean	$p - value$
100	*sdy*	6	9	6.94 ± 0.56	< 0.0001
100	*reg*	6	9	7.6 ± 0.61	
1000	*sdy*	5	8	6.65 ± 0.43	0.0007
1000	*reg*	5	9	7.01 ± 0.66	
10000	*sdy*	5	8	6.64 ± 0.45	0.2442
10000	*reg*	5	8	6.75 ± 0.43	

Table 1 The results from experiments with $s_r = 30$, $h_r = 1$ and $h_t = 1$. The last column shows the $p - value$ from a standard T-test between two subsequent rows.

$\mathbf{s_r}$	**spanning trees**	**min**	**max**	**mean**
10	100	15	22	18.7 ± 2.4
10	1000	14	20	16.8 ± 1.6
10	10000	13	18	15.6 ± 1.1
30	100	6	11	8.5 ± 0.9
30	1000	6	10	8.0 ± 0.7
30	10000	6	9	7.7 ± 0.6
50	100	6	11	8.0 ± 1.2
50	1000	6	11	7.7 ± 0.9
50	10000	6	11	7.7 ± 1.0
70	100	5	11	7.9 ± 1.0
70	1000	5	10	7.7 ± 0.9
70	10000	5	10	7.6 ± 1.0

Table 2 Impact of varying range and number of spanning trees on the sample map from Figure 12 with $h_p = 1.0$ and $h_t = 1.0$ from [20].

and 10000 randomly generated depth-first spanning trees, similar to [10]. Across all spanning trees the one leading to the best strategy was selected, i.e., the one needing the least amount of robots. The results are presented in Table 2. Only for the smallest sensing range $s_r = 10$ the difference in the number of spanning trees had an effect on the best strategy found, whereas for all other cases 100 spanning trees sufficed. This effect can be well explained by the fact that smaller sensing ranges lead to more vertices and thus larger graphs that in turn lead to more potential spanning trees that have to be considered.

Figure 13 shows the distribution of the number of robots across the 100 randomly generated graphs for 100 and 10000 spanning trees, and Figure 14 depicts the execution of one of the strategies computed on the *Sample map.*

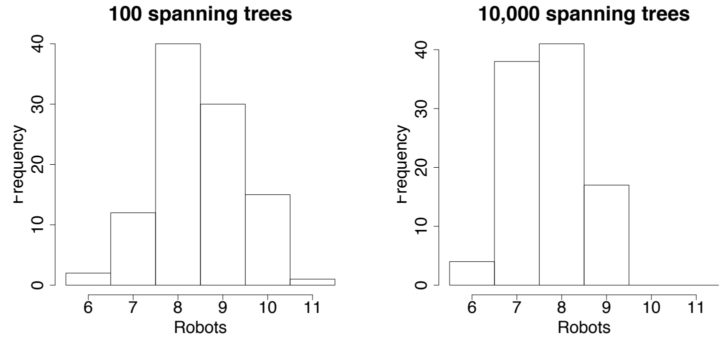


Fig. 13 Two histograms of the distributions of the number of robots needed for the 100 randomly generated graphs.

$\mathbf{s_r}$	$\mathbf{h_r}$	$\mathbf{h_t}$	**min**	**max**	**mean**
10	1	1	16	22	19.2 ± 1.7
30	1	1	9	17	12.1 ± 2.7
50	1	1	8	15	11.8 ± 2.1
70	1	1	8	16	11.7 ± 2.4
50	0.5	0.5	11	21	15.5 ± 3.4
50	0.5	1	10	17	12.8 ± 2.2
50	1	0.5	9	18	14.5 ± 2.6

Table 3 Impact of varying range, searcher and target height on the village map from Figure 12.

An increase of the sensing range from 10 to 30 reduced the number of needed robots significantly. However, any further increase had only marginal impact. Apparently a gain in sensing range is mitigated by the number of occlusions. With many occlusions an increase in sensing range is less likely leading to improvements.

The effect of varying sensing range was also confirmed by experiments conducted on the *Village* map shown in Figure 12(b). As shown by Table 3 varying the sensing range leads to a steep decrease in the number of robots from 10 to 30, whereas further changes had only marginal effects.

Since this map has a considerably elevation structure we also tested the effect of varying searcher and target heights h_r and h_t. A reduction of h_t from 1 to 0.5 required 9 instead of 8 for the same sensing range and $h_r = 1$. A reduction of h_r from 1 to 0.5 required 10 instead of 8 for the same sensing range and $h_t = 1$. Reducing both, h_t and h_r to 0.5 needed 11 instead of 8 robots.

This is likely due to the more complex structure of *Colorado*. In this case an increased sensing range does not yield a much larger detection set, but a detection set with a more complex boundary due to many more occlusions. This complex boundary leads to more edges in the graph. This is also shown by Figure 15 (b) that illustrates an increase of the number of edges for *Colorado* but not for the other maps as the sensing range increases. We will see in Section 8.2 that a modification of the graph generation algorithm can relax the effect of increasing graph complexity for complex environments.

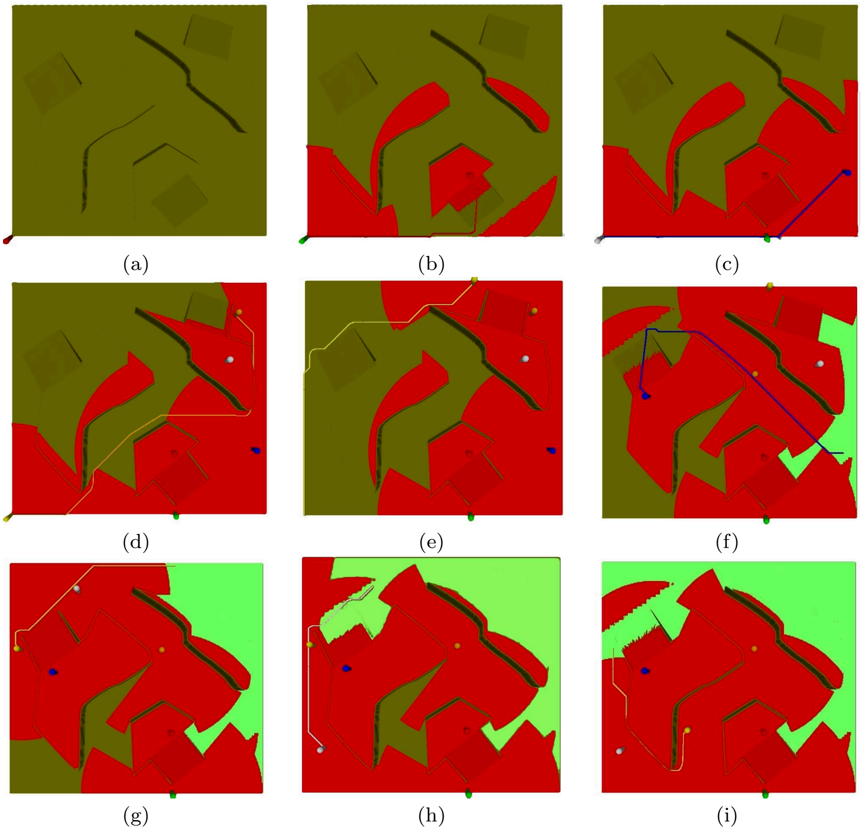


Fig. 14 A strategy for the sample map from Figure 12 with 6 robots. Detection sets are marked red and cleared areas not under observation are marked green. At step 0 on the upper left all robots are at their deployment location at the bottom left on the map. The pictures show steps $0, 1, 3, 5, 6, 7, 10$ and 12 from left to right and top to bottom. At each step the path of the last robot moving is drawn. At step 1 the first robot moves to a plateau and after step 5 the robots cleared half the map. In step 6 all 6 robots are required to avoid recontamination of the graph. In step 8 the first cleared but unobserved part of the environment appears until in step 12 the entire environment is cleared.

8.2 Hierarchical Sampling and Sparse Graphs

In this section we evaluate random sampling (Section 4.3) versus hierarchical sampling (Section 4.4), and shady edge generation (Section 4.5) versus sparse edge generation (Section 4.6). For this purpose more challenging maps shown in Figure 16 are taken for the experiments: (a) LiDAR data point cloud with 1 m resolution that was collected in response to the Haiti earthquake. The data was collected by the Center for Imaging Science at Rochester Institute of Technology (RIT) and Kucera International, and funded by the Global Facility for Disaster Recovery and Recovery (GFDRR) hosted at the World Bank [12]. (b-d) are DEM data from USGS [1] at 10 m resolution. (e) LiDAR data acquired on the campus of the University of Freiburg using a wheeled robot equipped with a SICK LMS laser range

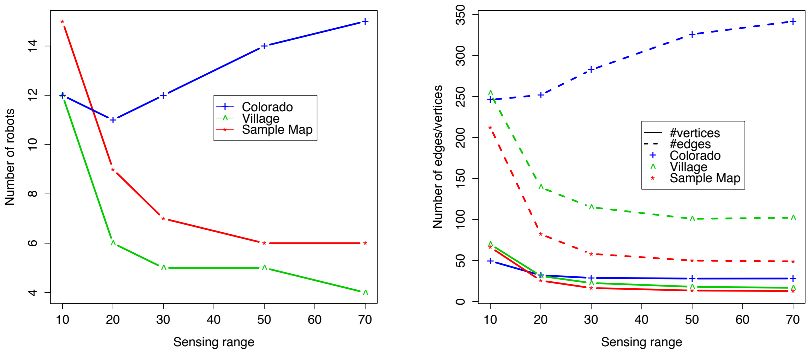


Fig. 15 (a) A plot of the number of robots needed for the best strategy at a given sensing range for all three maps. (b) A plot of the average number of vertices and edges for all three maps.

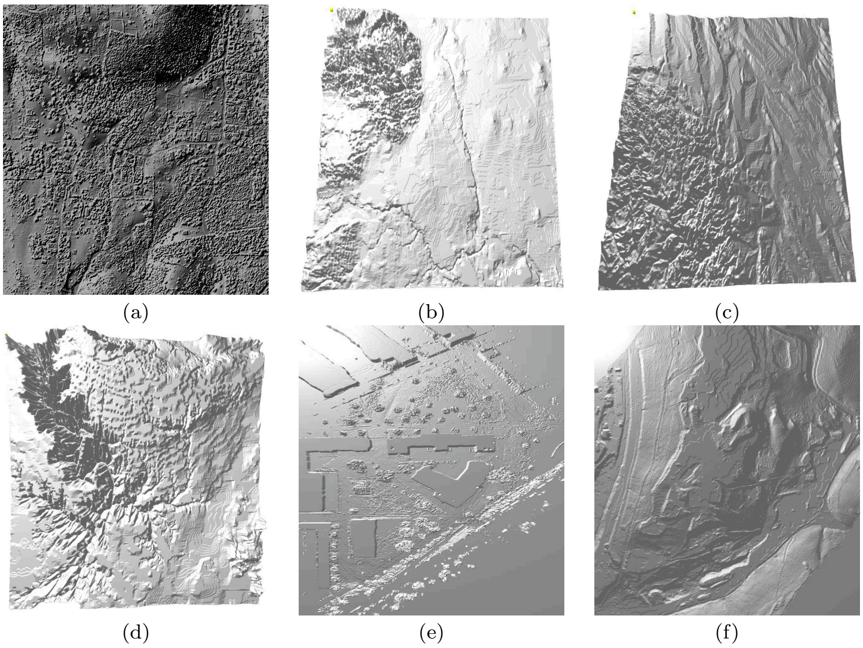

(a) (b) (c)
(d) (e) (f)

Fig. 16 Large-scale maps for testing hierarchical sampling: (a) DEM generated after the Haiti earthquake, (b-d) data from USGS, (b) Twinfalls (c) RapidCity (d) Grand Canyon, (e) map generated from LiDAR data of the Campus of the University of Freiburg, (f) DEM of the Gascola robot evaluation site.

scanner mounted on a pan-tilt unit [25]. The pan-tilt unit was moved to acquire a 360 degree view of the surrounding. (f) High resolution (1 m) DEM data of the Gascola robot evaluation site of the Carnegie Mellon University (Pittsburgh). All

map name	**variant**	**#V**	**#E**	**min**	**max**	**mean**
Gascola	*sdy+rnd*	159	408	27	36	31.7 ± 4.1
	sdy+hie	109	208	19	22	20.2 ± 0.6
	spa+rnd	102	240	13	18	13.9 ± 1.1
	spa+hie	62	130	7	8	7.8 ± 0.2
Freiburg	*sdy+rnd*	919	9276	N/A	N/A	N/A
	sdy+hie	470	2593	N/A	N/A	N/A
	spa+rnd	160	628	22	29	25.5 ± 2.8
	spa+hie	110	344	15	18	16.7 ± 0.4
Haiti	*sdy+rnd*	2297	64494	1017	1136	1087.8 ± 3348.7
	sdy+hie	1337	20939	493	562	524.3 ± 267.6
	spa+rnd	342	2236	73	91	79.0 ± 16.4
	spa+hie	144	634	24	29	27.4 ± 0.9
Grand Canyon	*sdy+rnd*	1554	10710	460	563	527.2 ± 596.7
	sdy+hie	784	3165	N/A	N/A	N/A
	spa+rnd	367	1636	51	73	61.4 ± 17.5
	spa+hie	208	741	25	34	31.9 ± 2.6
Rapid City	*sdy+rnd*	678	9693	359	400	378.0 ± 131.2
	sdy+hie	316	2185	129	152	143.9 ± 19.5
	spa+rnd	190	1258	41	59	48.9 ± 14.7
	spa+hie	89	446	22	24	22.9 ± 0.4
Twin Falls	*sdy+rnd*	696	7807	261	321	294.1 ± 224.0
	sdy+hie	341	1951	94	117	108.1 ± 20.2
	spa+rnd	150	762	24	35	29.7 ± 5.5
	spa+hie	75	288	14	15	14.2 ± 0.2

Table 4 Comparing random graph sampling (rnd) with hierarchical graph sampling (hie) and shady edge generation (sdy) with sparse edge generation (spa). The experiment has been carried out with $s_r = 500$, $h_r = 1.8$ and $h_t = 1.8$.

maps have been per-classified by the method describe in Section 4.1, based on the model of an *all terrain* robot.

Table 4 summarizes the results, where random sampling is denoted by *rnd*, hierarchical sampling by *hie*, shady edge generation by *sdy*, and sparse edge generation by *spa*. All presented results were averaged across 100 computations. The results clearly indicates that hierarchical sampling outperforms random sampling, as well as sparse edge generation outperforms shady edge generation, i.e., hierarchical sampling with sparse edge generation leads to strategies requiring the least amount of robots.

8.3 Field-Experiment Gascola

In this section we present results from applying the best strategy found with hierarchical sampling and sparse edge generation for the Gascola outdoor area. The elevation map of Gascola shown in Figure 16(f) has a resolution of $1m$ per pixel. The entire area of the site is approximately 700,000 m^2. The lowest point in the map is set to $0m$ elevation and the highest point is at $122m$. Gascola has a lot of seasonal shrubs and other vegetation that influence visibility and movement of agents. We therefore surveyed the terrain a week prior to the deployment of agents and added the annotations seen in Figure 17 (a). Collecting detailed elevation maps is a considerable efforts and these annotations allow us to accommodate short term changes in the terrain. Note that large-scale elevation maps containing vegetation and building structures can be obtained by airborne or satellite-based synthetic aperture radar (SAR) devices yielding resolutions of up to 10 centimeters. The Haiti map shown by Figure 16 (a), for example, has been generated after the earthquake in Haiti 2010 in order to analyse the extend of destruction of man-made and natural structures.

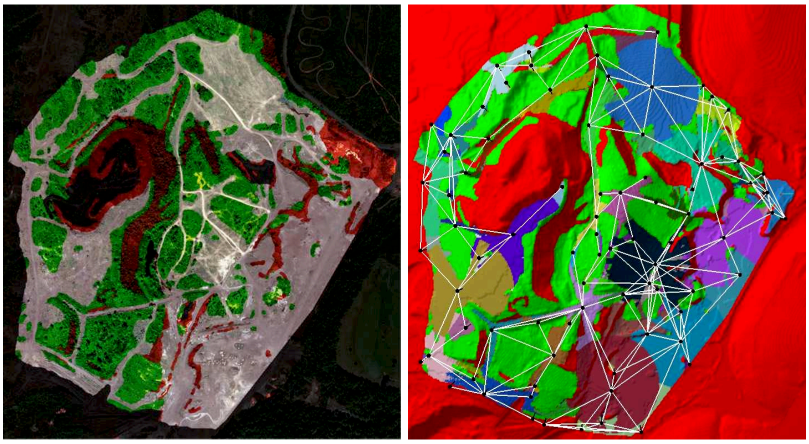
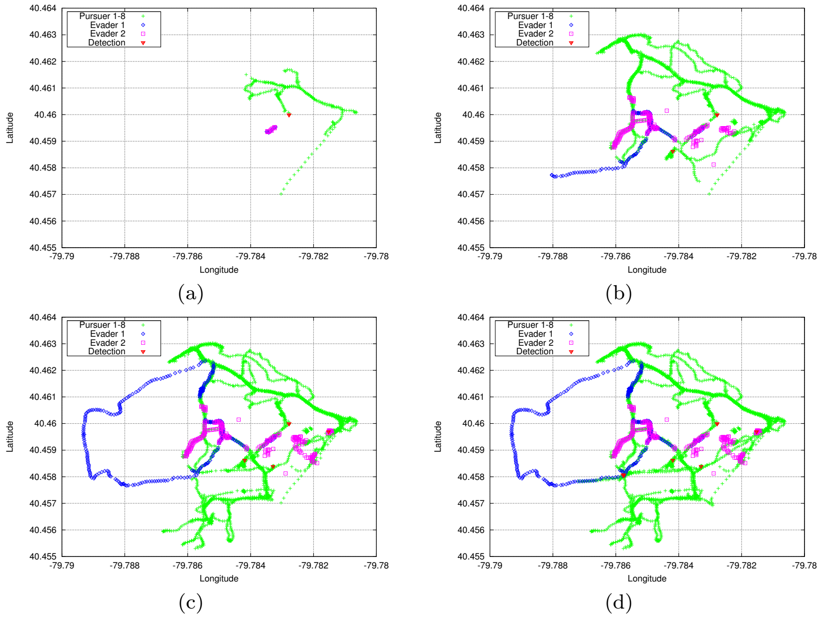


Fig. 17 (a) Map of the Gascola area outside of Pittsburgh with additional annotation: cluttered terrain, mostly shrubs and debris (green), steep areas that are none-traversable by agents (red), non-admissible areas defining the perimeter of the experiment (black). (b) Height map of the gascola area showing the generated graph for computing strategies (black vertices and green edges) with according detections sets (different colors).

As shown by Table 4 the hierarchical approach yields a significant reduction of the graph complexity particularly on cluttered maps while guaranteeing a full coverage of the terrain.

We selected a strategy requiring eight agents computed on the graph shown in Figure 17 (b), where detection sets associated with each vertex (and real world position) are shown with different colorings. These detection sets where uploaded to the mobile devices in order to inform agents about the detection sets they are responsible for. We then computed the execution time using our procedure from Section 6 yielding an assignment that takes 175 minutes to execute. In order to determine the impact of our procedure on execution time we compared it to 10,000 random assignments. These random assignments simply assign free agents randomly to new tasks at each step. Here we get a solutions with a mean execution time of 349.3056 ± 34.0350 minutes and with a maximum at 491.6365 and minimum at 236.4207. Hence the improvement is significant and can safe our searchers in the field in Gascola a whole hour of search time. Obviously, the problem deserves further study and experimentation on more maps. It should also be noted that instead of using an LBAP solution at each level we can solve the general assignment problem and thereby minimize the sum of all travel times instead of the maximum. This could be useful for applications in which energy conservation is more important and some of the execution time can be sacrificed.

All participants, eight searchers and two evaders, received a 15 minute instruction on how to use the application. The two evaders were given a head-start of another 15 minutes. They were instructed to make use of the available information on all searchers as best as possible to try to avoid being captured. Most agents were instantly able to follow the suggested paths and reach their locations. Two agents, however, had considerable difficulty at first to orient themselves and each one got lost once causing a delay of the execution but never leading to a breach between the boundary of contaminated and cleared space. After the first hour, however, all agents were comfortable following the instructions as the execution proceeded further. The experiment continued until the first IPads ran out of battery power. The searchers managed to execute two thirds of the entire strategy during this time and to catch every evader at least once.



(a) (b)

(c) (d)

Fig. 18 Snapshots of the GPS log from all searchers and evaders during the Gascola experiment after (a) 0.5 hour, (b) 1 hour, (c) 2 hours, and (d) 3 hours. Shown are all searcher trajectories (green), evader trajectories (purple and blue), and evader detections (red).

Figure 18 depicts snapshots of the GPS data recorded during the execution of the strategy. The purple evader was caught three times by three different agents attempting to move into the cleared areas undetected. The blue evader, however, managed to run behind the area controlled by one of the guards at the top of the map and successfully breached the perimeter. The GPS log clearly shows that the searcher in charge abandoned his area without instructions. This issue illustrates the necessity for thoroughly instructing the searchers when applying the system. The blue evader was, however, subsequently detected by another agent.

The main conclusions to draw from this field demonstration is foremost the feasibility of such an integrated system. Secondly, we observed that a team of human agents is by no means a homogeneous team. Each agent has different walk speeds and capabilities in following the instructions. Furthermore, the outdoor environment had changed due to rainfall, and some of the precomputed paths were in fact blocked. This had no effect on the guarantee of the strategy but did delay execution since alternative paths had to be found by the affected searchers. These two issues, heterogeneity and dynamic changes in the environment, clearly outline problems for further study.

9 Conclusion

We have proposed a novel and to our best knowledge first approach for guaranteed search in complex outdoor environments. Given an elevation map from an area including vegetation and man-made structures, as for example obtained by SAR devices, our system computes guaranteed schedules and navigation points from a deduced graph representation for teams of agent searching for targets. Although the presented method has no guarantee on optimality in terms of the number of needed searchers, several experiments have demonstrated a significant reduction of required agents, as well as the feasibility of the approach.

A novel graph structure has been presented that either generated randomly or based on hierarchical sampling captures visibility information arising in 2.5d problems represented by elevation maps. Several variants that utilize this information differently have been proposed. We have shown empirically that hierarchical sampling combined with sparse edge generation leads to the least amount of needed agents when computing contiguous strategies. Furthermore, we demonstrated the effects on strategies when changing the height of target and searchers, and the sensing range. In complex maps a larger sensing range can lead to worse strategies since the graph complexity increases due to unnecessary edges introduced from multiply overlapping detection sets. However, our approach allows it to identify empirically an appropriate sensing range leading to strategies requiring less robots for a particular map.

Despite the fact that the presented approach is based on heuristics we have demonstrated that it performs very well in complex real-world environments containing loops, occlusions, and significant height differences. The successful coordination of several human agents searching for evaders in a large outdoor area containing wild-growing shrubs, hills, and nested paths was demonstrated. The human team finally managed to capture all evaders when strictly following the guaranteed schedule computed beforehand.

The simplicity of our approach makes it readily applicable to a variety of domains. One nice property of the graph-based representation is that it facilitates heterogeneous teams, for example, mixed-initiative teams consisting of human and robot searchers. One direction of our future work is to integrate unmanned aerial vehicles (UAVs) into the search by computing detection sets individually for heterogenous agent types. With our current approach schedules are computed beforehand and cannot be changed during execution. The possibility to change schedules online can be an important feature when professional search teams are involved.

They might wish to modify strategies locally given their domain knowledge. Our future research will focus on an online-adjustable version of the proposed approach.

References

1. U.S. Geological Survey (USGS). `http://www.usgs.gov/`, 2010.
2. L. Barrière, P. Flocchini, P. Fraigniaud, and N. Santoro. Capture of an intruder by mobile agents. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 200–209, New York, NY, USA, 2002. ACM Press.
3. R. Borie, C. Tovey, and S. Koenig. Algorithms and complexity results for pursuit-evasion problems. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 59–66, 2009.
4. J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.
5. R.E. Burkard and E. Cela. Linear assignment problems and extensions. Technical report, Karl-Franzens-Univ. Graz & Techn. Univ. Graz, 1998.
6. P. Carraresi and G. Gallo. A multi-level bottleneck assignment approach to the bus drivers' rostering problem. *European Journal of Operational Research*, 16(2):163–173, 1984.
7. C. Dornhege and A. Kleiner. Behavior maps for online planning of obstacle negotiation and climbing on rough terrain. In *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots & Systems (IROS)*, pages 3005–3011, San Diego, California, 2007.
8. Charles Elkan. The paradoxical success of fuzzy logic. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 698–703, Menlo Park, California, 1993.
9. 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*, 9:471–494, 1999.
10. G. Hollinger, A. Kehagias, S. Singh, D. Ferguson, and S. Srinivasa. Anytime guaranteed search using spanning trees. Technical Report CMU-RI-TR-08-36, The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, August 2008.
11. Geoffrey Hollinger, Athanasios Kehagias, and Sanjiv Singh. Gsst: anytime guaranteed search. *Autonomous Robots*, 29(1):99 – 118, July 2010.
12. World Bank ImageCat Inc. RIT Haiti earthquake LiDAR. `http://opentopo.sdsc.edu/gridsphere/gridsphere?cid=datasets`, 2010.
13. R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38(4):325–340, 1987.
14. A. Kehagias, G. Hollinger, and A. Gelastopoulos. Searching the nodes of a graph: theory and algorithms. Technical Report ArXiv Repository 0905.3359 [cs.DM], Carnegie Mellon University, 2009.
15. A. Kleiner and C. Dornhege. Real-time localization and elevation mapping within urban search and rescue scenarios. *Journal of Field Robotics*, 24(8–9):723–745, 2007.
16. A. Kolling and S. Carpin. Extracting surveillance graphs from robot maps. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2323–2328, 2008.
17. A. Kolling and S. Carpin. Multi-robot surveillance: an improved algorithm for the Graph-Clear problem. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2360–2365, 2008.
18. A. Kolling and S. Carpin. On weighted edge-searching. Technical Report 01, School of Engineering, University of California, Merced, 2009.
19. A. Kolling and S. Carpin. Pursuit-evasion on trees by robot teams. *IEEE Transactions on Robotics*, 26(1):32–47, 2010.
20. A. Kolling, A. Kleiner, M. Lewis, and K. Sycara. Pursuit-evasion in 2.5d based on team-visibility. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2010. (accepted for publication).
21. S. Lazebnik. Visibility-based pursuit-evasion in three-dimensional environments. Technical report, University of Illinois at Urbana-Champaign, 2001.
22. M. Moors, T. Röhling, and D. Schulz. A probabilistic approach to coordinated multi-robot indoor surveillance. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3447–3452, 2005.

23. T.D. Parsons. Pursuit-evasion in a graph. In Y. Alavi and D. R. Lick, editors, *Theory and Applications of Graphs*, volume 642, pages 426–441. Springer Berlin / Heidelberg, 1976.
24. S. Sachs, S. Rajko, and S. M. LaValle. Visibility-based pursuit-evasion in an unknown planar environment. *The International Journal of Robotics Research*, 23(1):3–26, January 2004.
25. B. Steder. Freiburg campus LiDAR data. `http://ais.informatik.uni-freiburg.de/projects/datasets/fr360/`, 2010.
26. B. Tovar and S. M. LaValle. Visibility-based pursuit-evasion with bounded speed. In *Proceedings of the Workshop on Algorithmic Foundations of Robotics*, pages 475–489, 2006.