Hierarchical Visibility for Guaranteed Search in Large-Scale Outdoor Terrain

Alexander Kleiner, A. Kolling, M. Lewis and K. Sycara

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 Searching for moving targets in large environments 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 to guarantee the discovery of all targets. In this paper we present a self-contained solution to this problem in 2.5D real-world domains represented by digital elevation models (DEMs). We introduce hierarchical sampling on DEMs for selecting heuristically the close to minimal set of locations from which the entire surface of the DEM can be guarded. Locations are utilized to form a search graph on which search strategies for mobile agents are computed. For these strategies schedules are derived which include agent paths that are directly executable in the terrain. Presented experimental results demonstrate the performance of the method. 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. The field demonstration is the largest-scale implementation of a guaranteed search algorithm to date.

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

Searching for moving targets in large environments 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. These applications require the coordination of a team of searchers to guarantee the detection of all targets, a problem usually referred to as guaranteed search. Guaranteed search makes two worst-case assumptions: first, the motion model of targets is unknown and hence targets are assumed to travel with unbounded speed. Second, targets are acting adversarial and are omniscient. 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. Whenever the target is assumed to be omniscient and moving at unbounded speed most approaches make use of the concept of *contamination*. Contamination simply refers to the possibility of an unseen target being present at a location. The goal of the guaranteed search problem hence is to coordinate agents to clear environments from all contamination while using as few searchers as possible.

Demonstrations and applications of guaranteed search for real world scenarios face a number of challenges. For one much of the prior work on guaranteed search does not extend to large agent teams, and is limited to two-dimensional environments [39], or certain types of idealized sensors such as unlimited range target detection [18]. Overall, very little work has been done so far for 2.5D or 3D guaranteed search problems [33]. Graph-based approaches for guaranteed search with an emphasis on robotics [19, 29, 31] promise better scalability, but face another problem. Namely, the construction of an appropriate graph representation from a map, often given as a grid map computed from sensor data. Strategies computed on this graph then have to be translated back into assigned paths for agents with proper timing in their execution. These difficulties have so far prevented comprehensive applications of guaranteed search in the real world.

In this paper we present a comprehensive and novel computational solution for finding guaranteed search schedules in 2.5D real-world environments represented by elevation maps. This is carried out by extracting a search graph from the elevation map, computing a strategy requiring the fewest agents on this graph, and then computing for each step of the strategy an assignment of agents to vertices of the search graph to reduce travel time. Figure 1 gives an overview of the main parts of the system. Search graphs are extracted by a hierarchical sampling method that heuristically selects strategic locations with large detection sets. These detection sets are areas around a location on which targets are detectable by an agent at the respective location. The goal is to approximate the close to minimal set of locations from which the entire area of the DEM (digital elevation model) is covered. We compute overlaps between these detection sets in order to determine which regions can mutually guard their boundaries. The set of selected locations and edges extracted according to their overlaps, form the search graph on which clearing strategies are computed for the agent teams. For that purpose we introduce a variation of the Guaranteed Search with Spanning Trees (GSST) algorithm [19]. 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

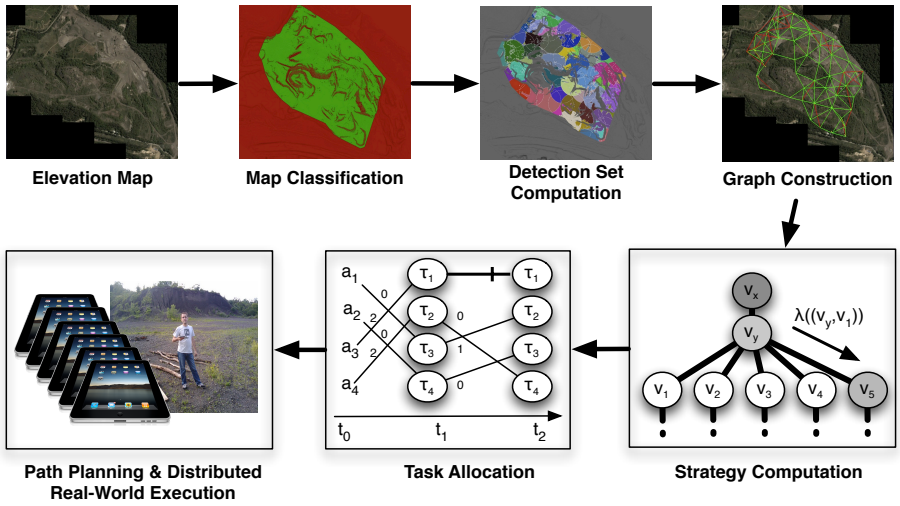


Fig. 1 Overview on the main parts of the system: From the classified elevation map a search graph is constructed from which a graph strategy and task assignments are computed. The task assignment assigns at each time step an area to be observed to each agent. Agents reach their target areas by path planning on the elevation map.

agents to locations in order to reduce the time needed to clear the environment. Locations are reached by path planning on the elevation map.

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 our experimental results, this enables the direct application 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. The Gascola robot training site is a wilderness area belonging to the Carnegie Mellon University in Pittsburgh and is mainly used for outdoor robotic experiments.

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

2 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 are always able to move to any other 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.

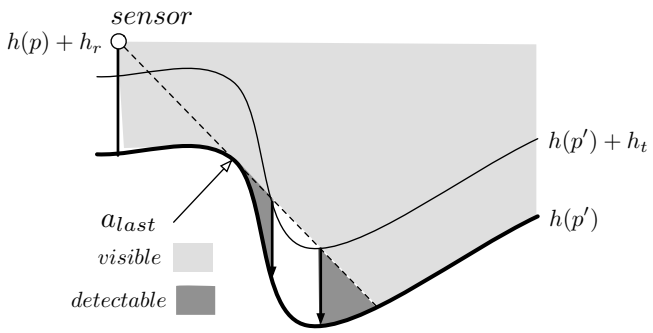


Fig. 2 An illustration how to compute detection sets for Algorithm 1.

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. A target at $p' \in H$ is detectable by a robot at p if at least one point on 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 2 for an illustration). Notice while H is considered as discretized into grid cells, height values and thus the z-component of the line segments are in $\mathbb{R}^+$.

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 for a 3D embedding of the elevation map. 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 detection sets $D(p)$ increases as well. With $h_t = 0$ we simply have targets visible whenever the ground on which they are located is visible. For practical applications this means that we can inform the user about the specific number of robots needed for a search given the minimal height of targets and searchers.

3 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+1} and then incrementing i to $i + 1$ until $\mathcal{E} \setminus \bigcup_{j=1}^{i} D(p_j)$ is the empty set. The locations p_i will each be identified with exactly one vertex $v_i \in V$ and we shall use v_i in the context of $\mathcal{G}$ and p_i when referring to v_i's location in $\mathcal{E}$. The two introduced methods are a random sampling procedure, first presented in [1], and a hierarchical 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 re-contamination.

3.1 Height Map Classification

Free space $\mathcal{E}$, representing the area in which agents can freely move, is constructed by an elevation map classification procedure. Elevation maps are widely available on the Internet as digital elevation models (DEMs), e.g. from USGS [48], at a resolution of up to 1 meter. Higher resolutions can be achieved by traversing the terrain with a mobile robot [25].

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 steeper slopes, and also stairs. These specific parameters are taken into account by the classifier described in the following. Notice that during our experiments the motion model of humans has continuously been used.

The procedure used for terrain classification is based on fuzzy features [13]. While simpler methods can also be used for classifying traversable terrain, such as computing the terrain slope for each map cell according to local neighbors, the framework presented in [13] has the advantage that it can easily be extended to more complex terrain features such as stairs, which finally allows us to deal with a wide range of motion models.

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 $SDown(x, a, b)$ (Equa-

tion 2), where a and b denote the desired range of the parameter.

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

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

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 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* = $SUp(\alpha_i, 10^\circ, 25^\circ) \cdot SDown(\alpha_i, 25^\circ, 40^\circ)$. The latter describes a trapezoid function in which, depending on the input angle, either SUp or $SDown$ returns the output value. 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 [14]. 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.

3.2 Detection Set Computation

In order to construct a graph $\mathcal{G}$ we need to be able to compute detection sets for locations $p \in \mathcal{E}$. The detection set for location p is computed by casting rays radially from p in that all cells within the maximum range are visited, and to determine for each ray which points in $\mathcal{E}$ are detectable as shown by Algorithm 1 (see Figure 2 for an illustration). The computation is carried out by generating with the Bresenham algorithm [6] 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 as these slopes are monotonic increasing.

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.
$\alpha_{last} \leftarrow -\infty$
for $p' on\, \mathcal{L}$ **do**
  $\alpha_{tmax} \leftarrow \frac{h(p')+h_t-h(p)-h_r}{\|p-p'\|}$ *// Compute slope between p' and p.*
  if $\alpha_{tmax} \geq \alpha_{last}$ **then**
    $D \leftarrow D \cup p'$ *// Add cell to detection only when slope monotonically increases.*
  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

Algorithm 2 *Random_Vertex_Construction*()

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

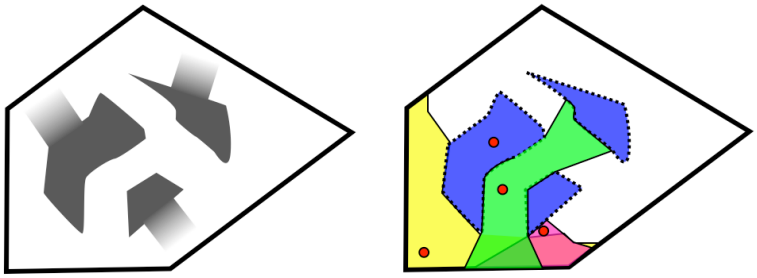
3.3 Random Vertex Sampling

In this section we present a first attempt to solve 2.5D pursuit-evasion by creating a graph by random sampling that captures the visibility information in the environment heuristically. The goal of this method is to sample a set of locations from which the entire area can be observed. The graph is directly embedded into the map and each vertex is associated to a location which can be used as a goal point for planning the motion of the robots assigned to it.

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+1}, $i = 1, 2, \ldots$ from $\mathcal{E} \setminus \bigcup_{j=1}^{i} D(p_i)$ and increment 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. Figure 3 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 $\mathcal{G} = (V, E)$.

3.4 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



(a) (b)

Fig. 3 (a) Height map representation where grey levels correspond to height values. (b) Sampled vertices (small circles) with overlapping detection sets (each depicted by a different color).

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 approach rather selects locations with good visibility, for example, mountain peaks or bell towers located on the map. The overall goal is to obtain the minimal set of locations needed to cover the entire area with the corresponding detection sets. This problem is generally known as the *set cover problem* and is one of Karp's 21 problems shown to be NP-complete [23]. In our setting already the computation of detection sets can be time consuming, especially, when many detection sets have to be computed in order to identify the largest one. To tackle this problem we present a greedy algorithm that selects the next best set by hierarchical sampling on low-resolution copies of the original map. As shown by Algorithm 3, 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 resolutions by assigning the maximum of the height values from the four corresponding cells on the lower level. Figure 4 depicts the generation of two low resolution maps at level 1 and 2 from the original map.

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 set $D(p_L)$. From these sets the location with the maximum detection set $p_L^{max} = \text{argmax}_{p_L} |D(p_L)|$ is selected. After locating 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}^{map} that corresponds to p_l^{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 cover-

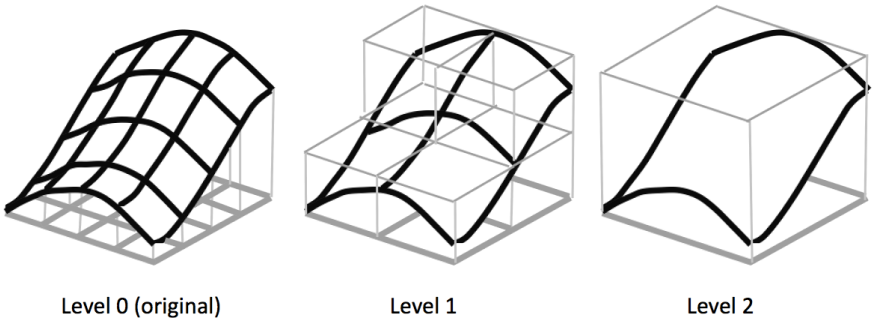


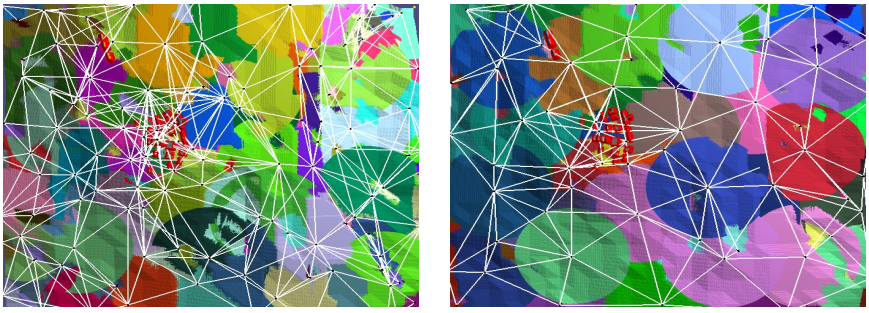
Fig. 4 Hierarchical simplification of elevation maps for computing detection sets. Level 0 represents the original map storing at each grid cell a height value. At higher levels height values are computed by combining the height values of the four grid cells of their predecessor level by the $max()$ operator.

Algorithm 3 *Hierarchical_Vertex_Construction*()

$i \leftarrow 0, V \leftarrow \emptyset$
while $\mathcal{E} \neq \emptyset$ **do**
  compute low-resolution copies $\mathcal{M} = (M_1, ..., M_L)$
  for all $p_L \in M_L$ **do**
    compute detection set $D(p_L)$
  end for
  $p_L^{max} \leftarrow \text{argmax}_{p_L \in M_L} |D(p_L)|$
  $l \leftarrow L$
  while $l \geq 0$ **do**
    $p_{l-1}^{map} \Leftarrow p_l^{max}$ // mapping from p_l^{max} to the next lower level cell
    $S_{l-1} \leftarrow \epsilon$-neighborhood p_{l-1}^{map}
    for all $p_{l-1} \in S_{l-1}$ **do**
      compute detection set $D(p_{l-1})$
    end for
    $p_l^{max} \leftarrow p_{l-1}^{max} \leftarrow \text{argmax}_{p_{l-1} \in S_{l-1}} |D(p_{l-1})|$
    $l \leftarrow l - 1$
  end while
  $\mathcal{E} \leftarrow \mathcal{E} \setminus D(p_0^{max})$
  $v_i \Leftarrow p_0^{max}$ // associating graph vertex v_i with map cell p_0^{max}
  $V \leftarrow V \cup v_i$, $i \leftarrow i + 1$
end while
Return V

ing p_l^{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 $p_{l-1}^{max} = \text{argmax}_{p_{l-1} \in S_{l-1}} |D(p_{l-1})|$. This procedure is continued until level 0 is reached and thus location p_0^{max} on the original map with maximal detection set is found. Then, $D(p_0^{max})$ is removed from $\mathcal{E}$. As shown by Algorithm 3, hierarchy $\mathcal{M}$ is recomputed from the reduced set $\mathcal{E}$ at each iteration of the outer *while* loop and thus also reflects modifications that occurred to $\mathcal{E}$. The hierarchical sampling continues until the entire map has been covered. Figure 5 depicts the result of random sampling versus hierarchical sampling on the *Village map*, which is an artificially generated map of a smaller village located on a hill (see Figure 13 on page 24 in the experimental results section). Hierar-

chical sampling leads to significantly simpler graph representations than random sampling while keeping the entire area covered.



(a) (b)

Fig. 5 Example graphs on the *Village map* generated (a) by random sampling, and (b) by hierarchical sampling.

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 7 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, which will be addressed in the subsequent two sections.

3.5 Shady Edge Computation

The edges of $\mathcal{G}$ should capture the neighborhood relationships between the detection sets $D(p_i)$ and thereby describe how detection sets can guard each others boundaries. In a 2D scenario the detection sets would be guaranteed to be connected, but in 2.5D they 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, regular and shady. 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 $G_{i,j} \neq \emptyset$ and there is no third vertex $v_{j'}$ whose guard region of v_i completely

Algorithm 4 *Shady_Edge_Construction*(V, P)

$E_r, E_s \leftarrow \emptyset, E_{r,dir}, E_{s,dir} \leftarrow \emptyset$
// Determine all directed regular and shady edges between all vertices.
for $i = 1$ *to* m **do**
  for $j = 1$ *to* m **do**
    $I \leftarrow \delta D(p_i) \cap D(p_j)$ *// Compute guard region.*
    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
// Transform the directed graph into an undirected one where regular edges dominate shady edges.
for $[v_i, v_j] \in E_{r,dir}$ **do**
  $E_r \leftarrow E_r \cup (v_i, v_j)$
  $E_{s,dir} \leftarrow E_{s,dir} \setminus \{[v_i, v_j], [v_j, v_i]\}$ *// Remove dominated shady edges*
end for
for $[v_i, v_j] \in E_{s,dir}$ **do**
  $E_s \leftarrow E_s \cup (v_i, v_j)$ *// Add shady edge*
end for
Return E_s, E_r

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 intuition behind creating two types of edges is as follows. 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 re-contamination if it can be guaranteed that no target can pass through $\delta D(p_i)$. We will show that this is satisfied when all vertices that are neighbors of v_i via regular edges are either clear or have a robot on them. The remaining edges that are not strictly necessary are shady edges which capture the remaining intersections between detection sets that are dominated by larger intersections from regular edges [1].

[1] One should note that this is a conservative approach and one could also use multiple shady edges to cover the same area of one regular edge. The problem here is that the graph-searching algorithms are not capable of such a generalized notion of preventing re-contamination. In order to accommodate this one would need to specify which sets of neighbors suffice to guard the boundary of a detection set and there may be multiple such sets.

3.6 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. More precisely, two vertices v_i, v_j receive an edge $[v_i, v_j]$ if and only if $\exists x \in G_{i,j}$ s.t. $x \notin D(v_{j'})$ for all $D(v_{j'})$ strictly larger than $D(v_j)$ or $D(v_i)$. 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 7 that this reduction has a positive impact on the strategy computation.

4 Strategy Computation

The search graph $\mathcal{G}$ constructed in Section 3 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 agents in order to clear $\mathcal{E}$ with as few agents as possible. 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 way point 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 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.

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 re-contamination 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, $\delta\left(\bigcup_{v' \in N_{unguarded}} \bar{D}(v')\right) \subset$

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

[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

$\bigcup_{v' \in N_{guarded}} D(v')$ 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 is clear then $\bar{D}(p)$ is clear for all v from $N_{unguarded}$ from step s. □

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. □

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.

4.1 Algorithm

Our resulting problem on $\mathcal{G}$ is very similar to the edge-searching problem as defined by Parson [38] and the variant with node contamination used in [19]. In fact, we can adapt algorithms from [2] and [19] to compute connected strategies without re-contamination. Recall that a connected strategy requires that all cleared vertices form a connected sub-tree. 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-connected strategies allow placement of agents far from the currently cleared area and hence can lead to long paths through contaminated areas. The algorithm from [2] was originally developed to handle a case in which multiple agents are required to clear a single vertex. However, it turns out not to be optimal for this purpose [11,30]. Yet, for the simpler unweighted case the resulting connected strategies are in fact optimal monotone and connected strategies on trees. In [24] 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.

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 7. 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 sub-tree 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 $m = degree(v_y) - 1$ 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 sub-trees 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 connected and without re-contamination. 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 sub-tree, i.e. v_1, is cleared. This is only a concern 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 sub-tree 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. Figure 6 depicts the label computation.

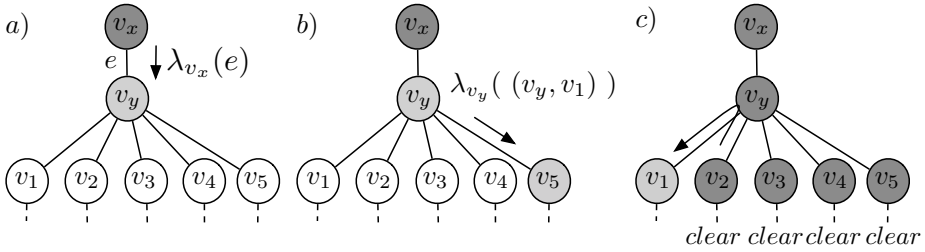


Fig. 6 An illustration of the label computation. Part a) shows an agent placed at v_y followed by part b) in which the team starts clearing the sub-trees until in part c) they enter the last sub-tree. White vertices are contaminated, light grey vertices are guarded, and dark grey vertices are cleared.

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 sub-trees 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 [19] 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. Hence, the cycle edges which were not part of the spanning tree can potentially lead to an increase in the number of agents required for the graph strategy since they force agents to remain at a vertex for longer. An agent can only leave a vertex in the graph once all its neighbors in the graph are cleared. In order to execute the strategy on the graph one might need additional agents. Fig. 7 illustrates this. In the worst case one will even need as many agents more as there are cycles

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

```
max_cost ← ∞ // number of needed agents
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 ← S_T, max_cost ← cost(S_G)
   end if
end for
Return best_strategy
```

in the graph. Finally, we select the converted graph strategy that requires the fewest robots. Algorithm 5 sketches this idea in pseudo code. Results presented in Section 7 confirm that this method works well in practice and with graphs constructed from real environments. Notice that other labeling procedures, such as random labels from [19], could also be used instead of the optimal labeling for the tree.

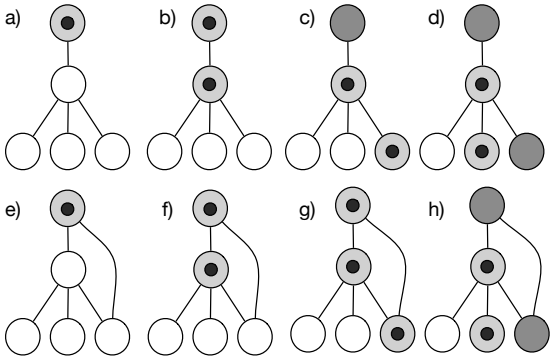


Fig. 7 An illustration of the conversion of tree strategies to graphs. Steps a) to d) show a strategy on the tree with agents as black dots and cleared vertices in grey. Steps e) to g) show the strategy on the corresponding graph with one additional cycle edge. This edge causes the agent on the top vertex to stay longer at its location until the neighbor is also cleared. This requires three instead of two agents.

5 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 connected 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 causing 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 [31] 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 thus 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 8 illustrates the new task sequence arising from a strategy.

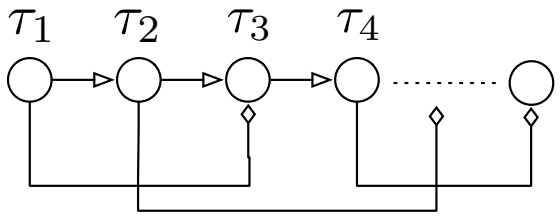


Fig. 8 A sequence $\tau_1, \tau_2, \tau_3, \ldots$ of tasks arising from a strategy, where circles denote tasks and edges towards a diamond denote task dependencies (i.e. mutual guarding constraints) which determine when agents on the associated locations can be removed. For example, at step 3 location 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.

[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.

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 7 this will be an A* planner) that returns the time it takes for an agent to travel between two locations in $\mathcal{E}$ written $\mathcal{T}(l, l')$. Let 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 9 shows three steps that finish at the same time, i.e. $t_3 = t_4 = t_5$.

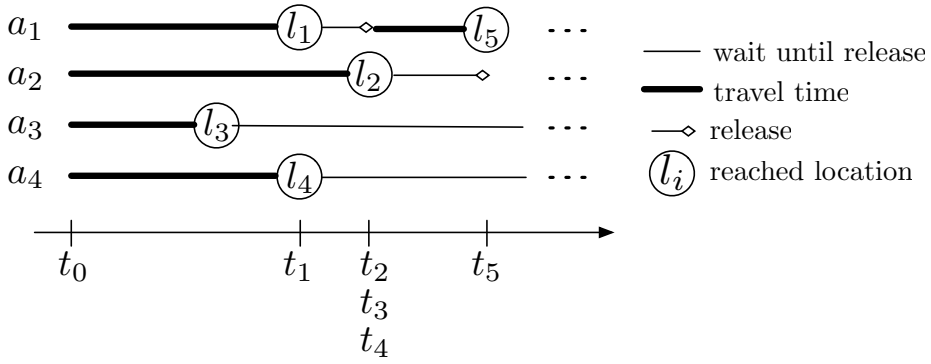


Fig. 9 Agents a_1, a_2, a_3 and a_4 move to locations l_1, l_2, l_3 and l_4 respectively. Step 1 is completed once a_1 reaches l_1. Other agents may already be at their assigned locations at this time. At step 2 agent a_1 is released and proceeds to l_5. Since a_3 and a_4 have already reached their task 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 all 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 be reached before new agents become available. In order to distinguish the notation we add a ˜ when referring to the compressed strategy.

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 $t_{arrival} - \tilde{t}_{i-1}$ 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 $|F_{i-1}| \geq |\tilde{T}_i \setminus \tilde{T}_{i-1}|$ 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 10 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 a different assignment for F_1 onto $\tilde{T}_2$. The assignments are also shown in Figure 11. 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 choose 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.

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 11 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 [9]. 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 7.4 we shall see that this already leads to a significant improvement over a naive approach, i.e. a random assignment, and enables the search of a large real environment with a reasonable search time. 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 [22]. Other methods are presented in the survey [8].

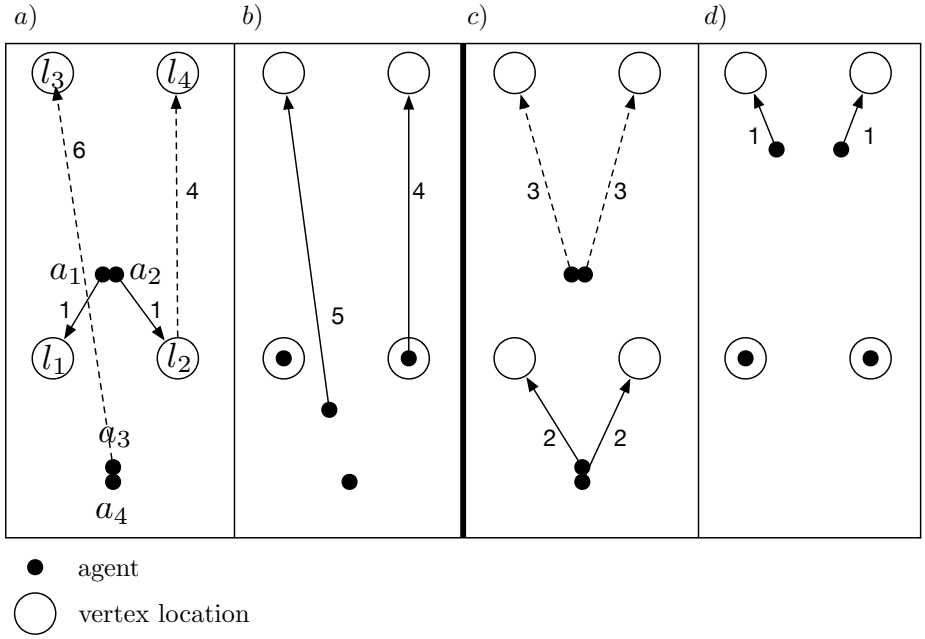


Fig. 10 Each part a)-d) shows four locations and agents executing part of a strategy with more vertices than shown here and requiring at least four robots. Part a) and b) show one assignment in which agents a_1 and a_2 move to l_1 and l_2, the optimal assignment to minimize $\tilde{t}_1$. After $\tilde{t}_1$ agent a_1 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 $\tilde{t}_2$.

However, this assignment is sub-optimal in terms of execution time since it ignores dependencies of subsequent steps in the schedule.

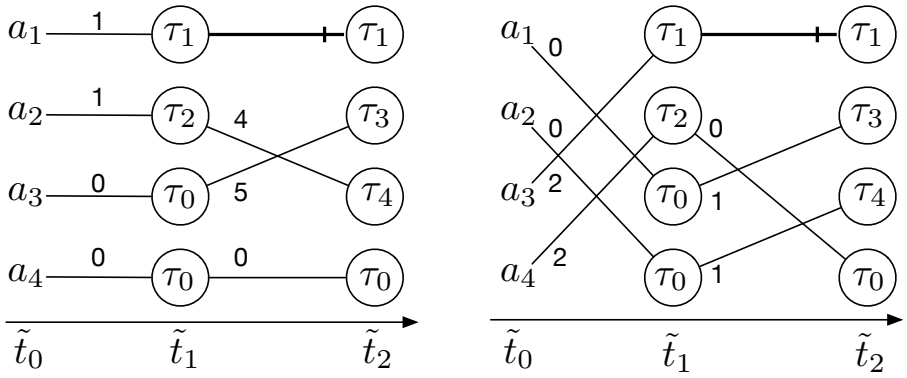


Fig. 11 Two task assignments visualized as graphs that correspond to Figure 10. The agent assigned to τ_1 has to remain there until release while τ_2 can be reassigned in the second level LBAP. The consecutive LBAP solution for both levels shown on the left is, however, not optimal for $\tilde{t}_2$.

6 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. As output, the system provides paths on elevation maps for guiding searchers through large outdoor environments. Using an annotated elevation map, a graph representation is constructed as described in Section 3 while taking into account additional obstructions for visibility due to cluttered terrain. On this graph, first, a strategy according to the algorithm presented in Section 4 and, second, a corresponding task assignment and schedule following the procedure described in Section 5 are computed. The schedule is then transferred to all agents for executing it online in the terrain.

For demonstrating our system we used human agents equipped with mobile devices (iPads) on which we programmed a custom Objective-C application. All devices were communicating via a mobile phone connection (third generation) and all data was logged at a central server. The interface of the application is shown in Figure 12. 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 their devices in real-time locations of all the searchers but also other evading agents. When 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 reached their assigned locations for an execution step, the system informed other agents about the progression by sending a message. Subsequently, agents were assigned to their new tasks automatically by the system.

7 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 7.1 we are investigating the performance of the graph construction based on random sampling on three example maps. We further investigate the question of whether shady edges can be ignored or whether there are instances in practice for which considering shady edges leads to better connected strategies. Second, in Section 7.3 the impact of hierarchical sampling and sparse edges is examined on large-scale maps. Finally, in Section 7.4 results from deploying the system in the field, the *Gascola* outdoor area around Pittsburgh, are presented.

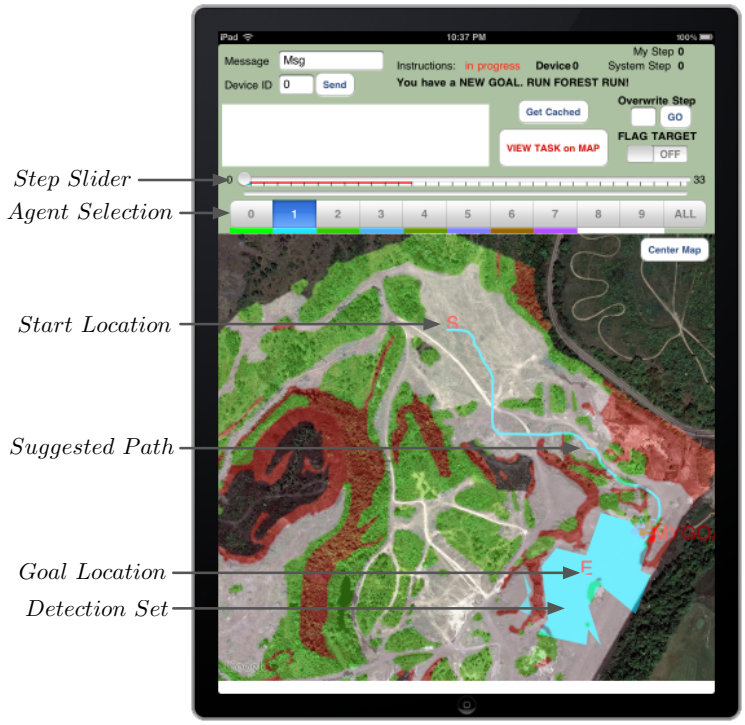


Fig. 12 The iPad interface for all agents. Additionally each agent receives real-time locations and plans of other searchers that are directly displayed on the device. Evading agents receive the same information in addition to information about other evaders.

7.1 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 [1]. Recall the definition of regular edges, E_r, and shady edges, E_s, from Section 3.5. Both types of edges capture some aspects of the neighborhood relations between detection sets. From our formulation of the graph model in Section 4 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 strictly necessary for avoiding re-contamination in $\mathcal{E}$. But visual proximity can be important for connected strategies. The advantage of connected 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 connected strategies can increase and possibly include one that is better than connected strategies based only on regular edges. For non-connected 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. The question how to treat shady edges was first raised in [1] and investigated experimentally. Combining our work from Section 4 with Algorithm 5 we now resolve the issue more precisely.

Consider Algorithm 5 from Section 4. 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 connectedness requirement can only have an impact on T since the strategy is already required to be connected on T (which translates to connectedness 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 [1] 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. For both variants we compute strategies on the spanning tree T as presented in Section 4 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 [1] did in fact address the question whether *sdy* or *reg* leads to generally better strategies. In [1] this was denoted as a *bias* in the spanning tree generation and in what follows we will adapt the relevant results from [1] to our context. The experiments were carried out by randomly sampling vertices on three maps depicted in Figure 13. 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 the same 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.

Table 1 summarizes the results for the comparison between *reg* and *sdy* for different number of spanning trees used for the computation of strategies. A standard T-test with associated p-value was conducted to compare the best strategies for both variants across the 100 randomly sampled graphs on the *Sample map*. When fewer, i.e. 100, spanning trees were generated the two variants performed significantly different $p < 0.0001$. The average number of robots needed for *reg* was 6.94 ± 0.56 while *sdy* required 7.6 ± 0.61. Generating more spanning trees reduced this difference and the means became more similar and with 10,000 spanning trees they are statistically not significantly different, $p = 0.2442$. The best strategy across all graphs was always identical and the choice of variant had no

(a) (b) (c)

Fig. 13 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	*reg*	6	9	6.94 ± 0.56	< 0.0001
100	*sdy*	6	9	7.6 ± 0.61	
1,000	*reg*	5	8	6.65 ± 0.43	0.0007
1,000	*sdy*	5	9	7.01 ± 0.66	
10,000	*reg*	5	8	6.64 ± 0.45	0.2442
10,000	*sdy*	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.

impact on the number of agents ultimately required to clear the graph. These experiments indicate that if one can generate large numbers of spanning trees one can safely ignore the difference between *reg* and *sdy*. When fewer spanning trees are generated the bias towards regular edges in *reg* allows the algorithm to test better spanning trees earlier. In general, the larger the graph the more spanning trees would be required to reach adequate performance and in this case the bias towards regular edges could lead to improvements. If one tests many or possibly all spanning trees then *sdy* performs equally well with the added possibility that it can find good spanning trees amongst those with shady edges.

Further experiments from [1], 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 13(a). For each randomly sampled graph the best strategy was computed based on 100, 1,000, and 10,000 randomly generated depth-first spanning trees, similar to [19]. 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 14 depicts the execution of a strategy computed on the *Sample map*.

s_r	spanning trees	min	max	mean
10	100	15	22	18.7 ± 2.4
10	1,000	14	20	16.8 ± 1.6
10	10,000	13	18	15.6 ± 1.1
30	100	6	11	8.5 ± 0.9
30	1,000	6	10	8.0 ± 0.7
30	10,000	6	9	7.7 ± 0.6
50	100	6	11	8.0 ± 1.2
50	1,000	6	11	7.7 ± 0.9
50	10,000	6	11	7.7 ± 1.0
70	100	5	11	7.9 ± 1.0
70	1,000	5	10	7.7 ± 0.9
70	10,000	5	10	7.6 ± 1.0

Table 2 Impact of varying range and number of spanning trees on the *Sample map* from Figure 13 with $h_p = 1.0$ and $h_t = 1.0$ from [1].

s_r	h_r	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 13.

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 13(b). As shown by Table 3 varying the sensing range from 10 to 30 leads to a steep decrease in the number of robots, whereas further changes had only marginal effects. Since this map has a considerable 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 shows that the effect of changing h_r can be quite different from the effect of changing h_t, i.e. these two values are not symmetric even though an increase in either will lead to larger detection sets.

The results of tests on all three maps with sensing ranges from 10 to 70 are presented in Figure 15 (a). Most notably, as the sensing range increases in maps *Sample map* and *Village* the number of robots decreases, but in map *Colorado* it first improves slightly and then gets worse. This is likely due to the more complex structure of *Colorado*. In this case an increased sensing range does not yield a

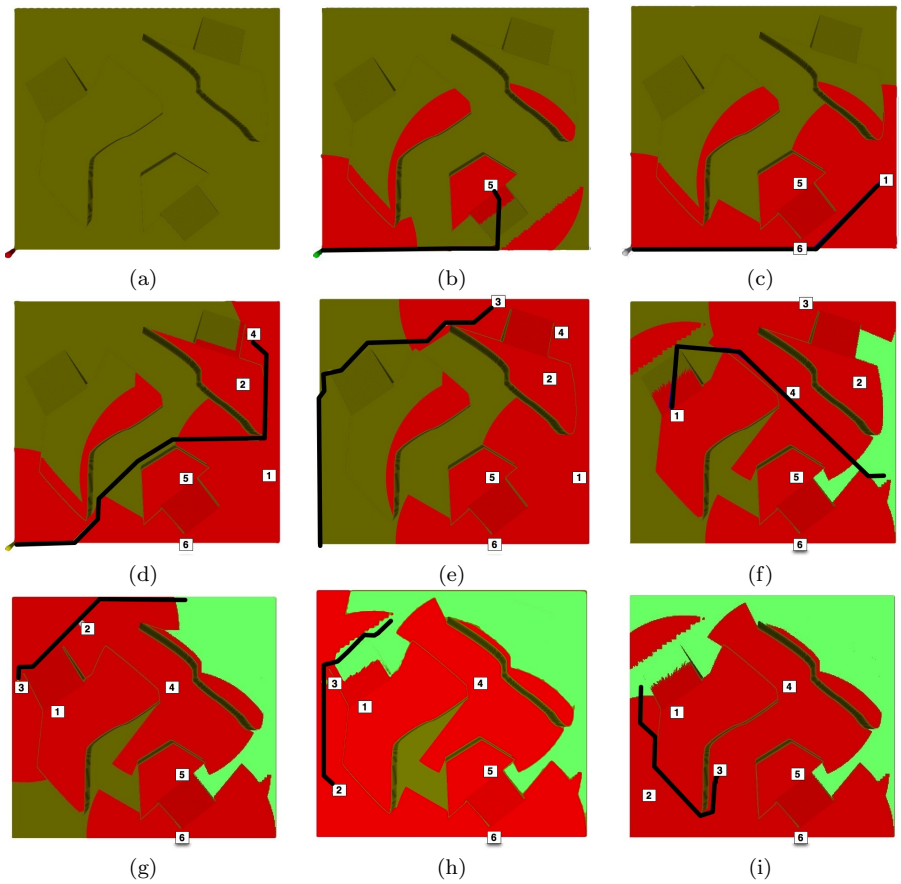


Fig. 14 A strategy for the *Sample map* from Figure 13 with 6 robots. Detection sets are marked red and cleared areas not under observation are marked green. At step 0 on the upper left all robots are at their deployment location at the bottom left on the map. The pictures show steps $0, 1, 3, 5, 6, 8, 10, 11$ 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 (b) the first robot moves to a plateau and after step 5 (d) the robots cleared half the map. In step 6 (e) all 6 robots are required to avoid re-contamination of the graph. In step 8 (f) the first cleared but unobserved part of the environment appears until in step 12 (i) the entire environment is cleared.

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. These edges together with the constraints of no re-contamination and connectedness make clearing the environment more difficult. This is supported by Figure 15 (b) that shows 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 7.3 that a modification of the graph generation algorithm can relax the effect of increasing graph complexity for complex environments.

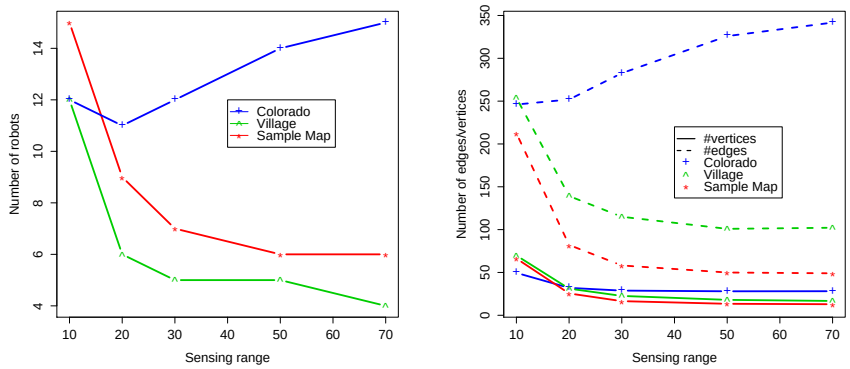


Fig. 15 (a) A plot of the number of robots needed for the best strategy at a given sensing range for all three maps. (b) A plot of the average number of vertices and edges for all three maps.

7.2 Comparing the Heuristic Solution with the Optimal Solution

One problem with evaluating the quality of the constructed graphs is that the graph strategies we compute by considering random spanning trees are not necessarily optimal. But as demonstrated in [19] on very small graphs it is possible to enumerate all spanning trees and perform an exhaustive computation of all strategies in order to determine the optimal one. This allows for the comparison of the quality of graphs directly by considering the optimum strategy. Since in the worst case, for complete graphs with n vertices, the number of spanning trees is n^{n-2} we can only perform such a comparison on very small graphs. For this purpose we created two low-resolution versions of the *Sample map* (53 X 48 pixels) and the *Village map* (41 X 46 pixels). On these low resolution maps the generated graphs generally have less than ten vertices. Notice that much of the structure of the *Village map*, i.e. the actual village in the center, is lost at this low resolution and the map becomes much simpler.

In Figure 16 the distributions of the costs of strategies on 10 million graphs constructed with random sampling is shown for both maps. The least amount of agents needed for the low-resolution versions of the *Sample map* (Figure 16(a)) and *Village map* (Figure 16(b)) are 3 and 2, respectively. When constructing a graph with hierarchical sampling (once) and computing the optimal strategy, the exact same result has been returned. For the *Village map* it is quite unlikely (2.8%) to find a graph yielding the best solution when deploying random sampling and thus the hierarchical construction indeed finds graphs that yield good strategies.

Now a second question is whether the strategies computed by considering only a limited set of spanning trees are close to the optimum strategy for each graph. This question was also addressed in [19] and the test therein revealed that the optimal solution was found in fact when sampling only ten spanning trees for a graph with 3604 possible spanning trees. In our case, the graph constructed with hierarchical

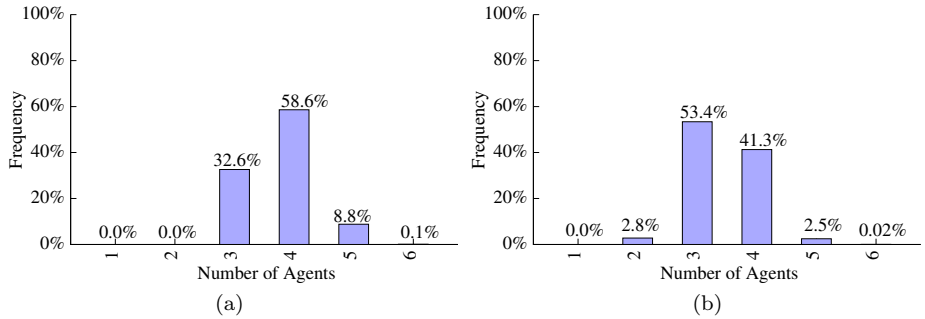


Fig. 16 Distribution agents needed for clearing the graph of solutions returned by random sampling when evaluating the entire set of spanning trees on low-resolution versions of the (a) *Sample map* and (b) *Village map*. The x-axis denotes the number of agents needed for clearing the graphs.

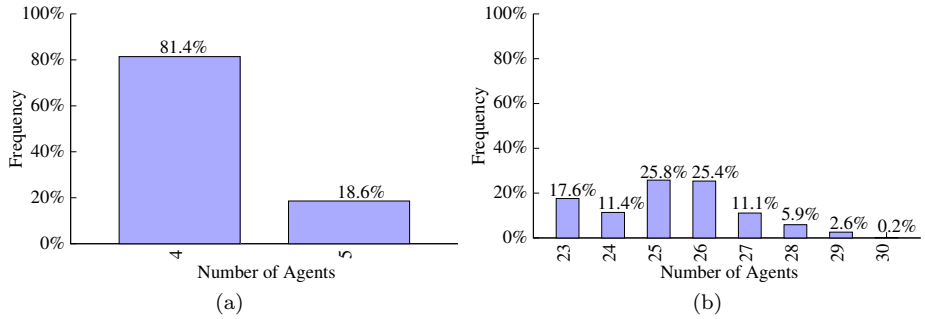


Fig. 17 Distribution of the number of agents needed for graph clearing when executing different strategies from enumerated spanning trees: on the full-resolution versions of the (a) *Sample map* and (b) *Village map*.

sampling on the full resolution version of the *Sample map* leads to a graph with 9596856 ($9.5 * 10^6$) spanning trees, computed using Kirchoff's theorem [47]. The distribution of the best strategies for each spanning tree is shown in Figure 17(a). All spanning trees lead to a strategy using either 4 or 5 agents. Hence, randomly sampling in this space of spanning trees is very likely going to return the optimal solution of 4 agents. Note that the full resolution version has greater detail and hence needs one more agent than the low resolution version.

For the full-resolution version of the *Village map* the graph constructed with hierarchical sampling has 10^{72} spanning trees. This renders the computation of the optimal solution impossible. Figure 17(b) depicts the distribution of the costs after enumerating the first 20 million spanning trees. Note that since we are using Char's algorithm to enumerate spanning trees as in [19] we are not guaranteed a representative sample of spanning trees. Despite the fact that the graph of the full resolution *Village map* is considerably more complex due to the structures inside the village we still get a relatively balanced distribution and sampling 100,000 spanning trees will very likely return at least one with the same number of agents than the best out of 20 million.

7.3 Hierarchical Sampling and Sparse Graphs

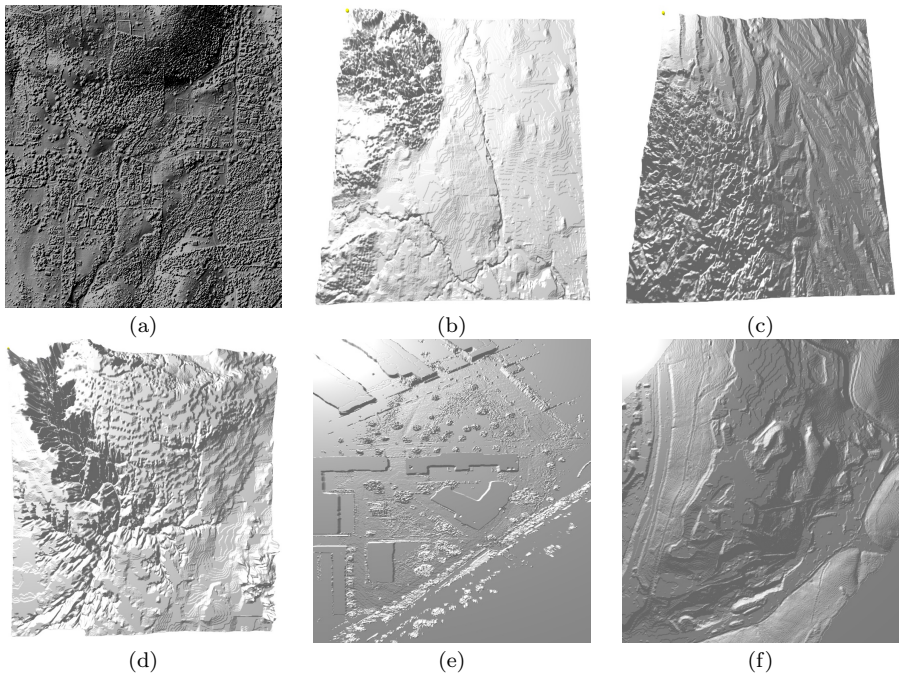


Fig. 18 Large-scale maps for testing hierarchical sampling: (a) DEM generated after the Haiti earthquake, (b-d) data from USGS, (b) Twinfalls (c) RapidCity (d) Grand Canyon, (e) map generated from LiDAR data of the Campus of the University of Freiburg, (f) DEM of the Gascola robot evaluation site.

In this section we evaluate random sampling (Section 3.3) versus hierarchical sampling (Section 3.4), and edge generation from Section 3.5, using the *reg* variant that only considers regular edges in the spanning tree, versus sparse edge generation (Section 3.6). Strategies on the graphs were computed by considering each time $100,000$ random spanning trees.

For this evaluation more challenging maps shown in Figure 18 were 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 [20]. (b-d) are DEM data from USGS [48] 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 scanner mounted on a pan-tilt unit [43]. The pan-tilt unit was used to acquire a 360 degree view of the surroundings. (f) High resolution (1 m) DEM data of the Gascola robot evaluation site of the Carnegie Mellon University (Pittsburgh). All maps have been

map name	variant	#V	#E	min	max	mean
Gascola	*spa+hie*	53	111	7	8	7.8 ± 0.4
	reg+hie	53	114	9	10	9.9 ± 0.4
	spa+rnd	157	359	12	18	14.9 ± 1.4
	reg+rnd	157	580	29	39	33.4 ± 2.1
Freiburg	*spa+hie*	110	344	15	18	16.5 ± 0.8
	reg+hie	110	455	22	25	24.2 ± 0.7
	spa+rnd	924	2952	N/A	N/A	N/A
	reg+rnd	921	16655	N/A	N/A	N/A
Haiti	*spa+hie*	193	892	30	37	34.5 ± 1.4
	reg+hie	193	1222	43	53	49.2 ± 2.0
	spa+rnd	2315	12503	299	367	331.0 ± 16.5
	reg+rnd	2318	71985	1115	1429	1292.6 ± 84.5
Grand Canyon	*spa+hie*	241	872	33	39	35.9 ± 1.4
	reg+hie	241	1140	46	55	50.7 ± 2.4
	spa+rnd	1551	6297	191	256	222.1 ± 11.6
	reg+rnd	1554	22376	514	623	579.8 ± 21.7
Rapid City	*spa+hie*	89	446	22	24	23.0 ± 0.6
	reg+hie	89	747	35	39	37.1 ± 0.8
	spa+rnd	681	3485	80	104	92.0 ± 5.4
	reg+rnd	684	19842	347	428	393.1 ± 19.7
Twin Falls	*spa+hie*	75	288	13	15	14.0 ± 0.5
	reg+hie	75	418	21	24	22.9 ± 0.6
	spa+rnd	694	2879	55	73	64.8 ± 4.5
	reg+rnd	698	12595	271	339	294.4 ± 14.7

Table 4 Comparing the „number of agents needed" when using random graph sampling (rnd), hierarchical graph sampling (hie), regular edge generation (reg), and sparse edge generation (spa). The experiment has been carried out with $s_r = 100$, $h_r = 1.8$ and $h_t = 1.8$.

pre-classified by the method describe in Section 3.1, 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*, regular edge generation by *reg*, and sparse edge generation by *spa*. All presented results for random graph sampling were averaged across 100 computations. The results clearly indicates that hierarchical sampling outperforms random sampling, as well as sparse edge generation outperforms regular edge generation, i.e., hierarchical sampling with sparse edge generation leads to strategies requiring the least amount of robots.

Figure 19 depicts the computation times required for graph sampling and strategy computation on the tested maps with sparse edge generation and regular edge generation on a *IntelCore(TM)2 Quad CPU @ 3.00GHz with 4GB RAM*. Depicted results are averaged over 100 runs. The computation of strategies requires notably more time on regular edge graphs than on sparse edge graphs. Hierarchical sampling requires constantly more time than random sampling, however, facilitates on every map a faster computation of the strategy once a graph has been found. For some maps, the total computation time is higher when using hierarchical sampling. However, this increase seems to be acceptable when considering the strategy improvement documented by Table 4.

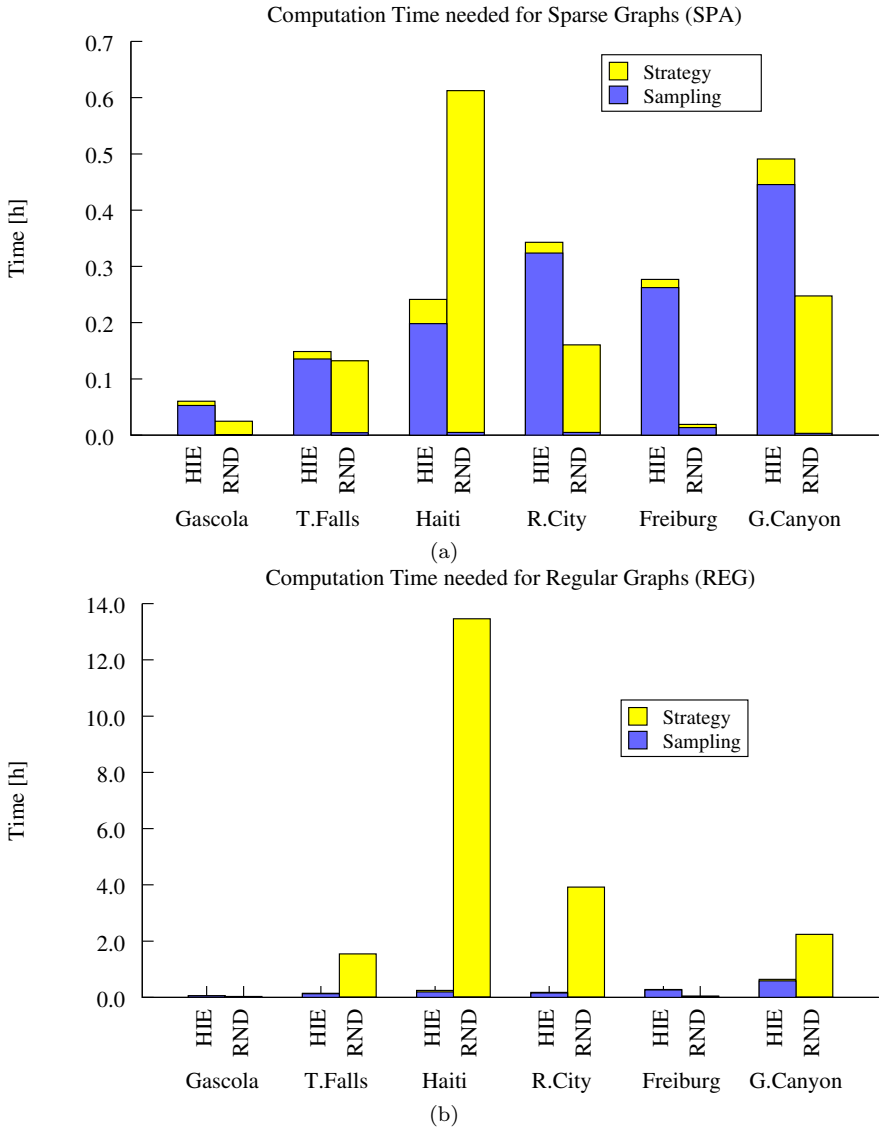
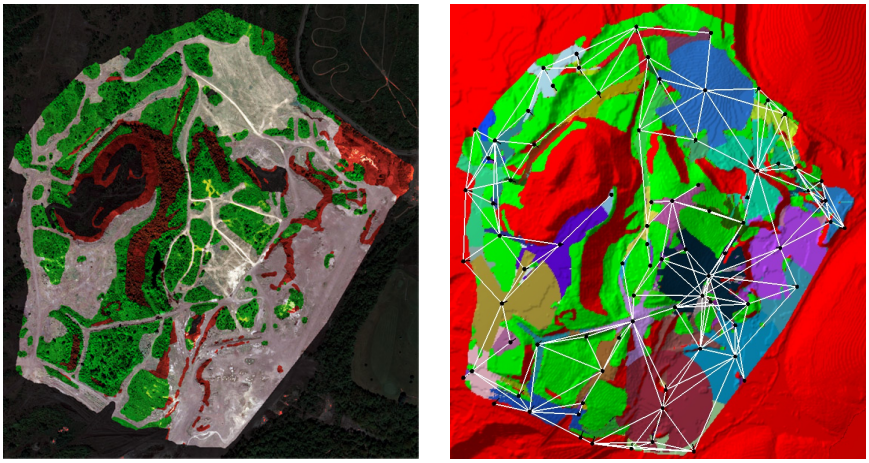


Fig. 19 Comparing computation times of random sampling (RND) and hierarchical sampling (HIE) on the tested maps. While the computation times for hierarchical sampling increase with increasing complexity of the map, strategies are faster computed due to simpler structures of the sampled graphs.

7.4 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 18(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 20 (a). Collecting detailed elevation maps is a considerable effort 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 18 (a), for example, has been generated after the earthquake in Haiti 2010 in order to analyze the extend of destruction of man-made and natural structures.



(a) (b)

Fig. 20 (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 white edges) with according detection sets (different colors).

We selected a strategy requiring eight agents computed on the graph shown in Figure 20 (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 areas they are responsible for. We then computed the execution time using our procedure from Section 5, using the algorithm from [22], yielding an assignment that takes 175 minutes to execute with a walking speed of 1.1 meter per second (approximately 4 km/h). 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 solution with a mean execution time of 349.3 ± 34.0 minutes and with a maximum at 491.6 and minimum at 236.4. Hence the improvement is significant and can save our searchers in the field in Gascola a whole hour of search time. Obviously, the problem de-

serves 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.

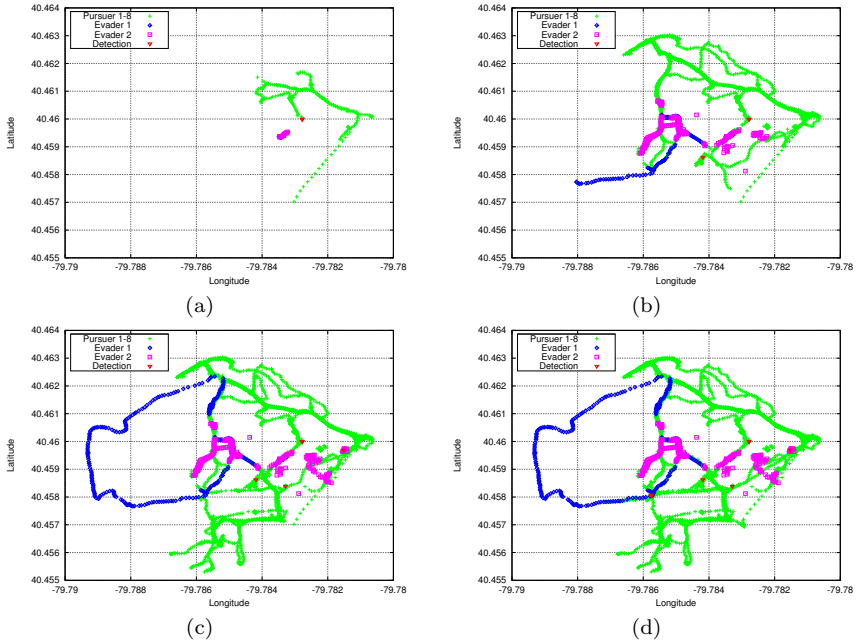


Fig. 21 Snapshots of the GPS log from all searchers and evaders during the Gascola experiment after (a) 0.5 hour, (b) 1 hour, (c) 2 hours, and (d) 3 hours. Shown are all searcher trajectories (green), evader trajectories (purple and blue), and evader detections (red).

Figure 21 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.

8 Related Work

Searching for moving targets is a general type of problem that has been considered in a variety of research areas ranging from robotics, control theory, sensor networks, computational geometry, up to graph theory. Most these areas emphasize different aspects of the problem and make different assumptions on the environment, targets, robots and sensors. For example, target locations may be unknown, known, sensed locally, or predictable. Target behavior may be probabilistic, deterministic, or adversarial with the target either having no knowledge, sensing locally, or being omniscient. Similarly, sensors can be local or global, have limited or unlimited range, be noisy or perfect. Environments may be graphs, 2d obstacle grids, polygons, or elevation maps. But the goal is generally to determine coordinated motion strategies for one or more agents that guarantees the detection of all targets in the given environment. This sets the problem apart from other popular areas such as the art gallery problem and its many variants [41], as well as coverage problems [7, 10]. In the following we shall present a small selection of this related work with particular emphasis on approaches that share our assumptions with regard to target behavior, i.e. unbounded speed, adversarial, and omniscient.

One of the more prominent areas relating to moving target search is known as visibility-based pursuit-evasion. This type of problem, with early variants proposed in [45], considers the detection of a worst-case target that moves arbitrarily fast and is omniscient with an unlimited range sensors. These worst-case targets are generally represented by contamination, i.e. the possibility of a target being present at a location.

Much of the work on visibility-based pursuit-evasion is concerned with two-dimensional environments. In [18] it was shown that determining the number of robots needed for an environment is NP-hard. For single robots, however, a number of algorithms have been developed. Most notably, [39] presents an on-line algorithm for a point pursuer moving in an unknown, simply-connected, piecewise-smooth planar environment. The algorithm is capable of searching for targets with a robot equipped with a sensor that only measured depth-discontinuities and could move only according to simple motion primitives (wall-following and along depth-discontinuities). Control was assumed to be imperfect with bounded errors. Similar to prior work their approach builds a navigation graph based on its motion prim-

itives and critical events. These critical events are determined from the geometry of the environment and capture how the space visible by the robot changes as it moves. These critical events occur when the sensed gaps appear, disappear, merge, or split. Since each gap represents one connected and currently not visible part of the environment the environment can be considered cleared once all gaps are known to be cleared as well. Gaps are cleared whenever they appear, with the exception of all gaps at the beginning which are contaminated. Cleared gaps that merge remain clear and contaminated gap merging with cleared gaps result in a contaminated gap. In order to guarantee the detection of the target, the robot has to find a motion strategy so that every starting gap disappears at least once and only cleared gaps remain. To compute such a strategy the algorithm superimposes information states, i.e. whether gaps are contaminated or not, on the navigation graph and then searches for a strategy that leads to an all cleared state.

Another approach was developed by Tovar et al. [46], who considered 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 re-contamination can be modeled, i.e. previously visible regions are not instantaneously recontaminated, but depending on the distance to the contaminated regions, they will only recontaminate after a certain time has passed. There are still open questions in this direction when considering further assumptions on the motion of the evader. A difficulty of the approach is to describe how the recontaminated regions, so called fans, evolve with time.

The main problem with visibility based approaches is that they are difficult to extend to large robot teams and large environments, especially complex 2.5D or 3D environments. An elaborate algorithm for two searchers is available in [42] and coordination becomes exceedingly difficult with larger teams. A report by Lazebnik [33] discusses the challenges of extending the ideas from 2D visibility-based pursuit-evasion to 3D. Especially the concept of critical events becomes difficult to tackle. One reason for this is that the unseen parts of the environment, i.e. shadow spaces, are not connected anymore, even in simply-connected environments. Another reason is that while the critical events in 2D can be determined via computing bi-tangents and inflectional tangents in 3D one needs to resort to more complex structures such as the aspect graph [26]. Constructing this graph for polyhedral scenes with n faces is $\Theta(n^9)$. While certainly interesting from a theoretical perspective, much further work is required to allow us to employ such an approach in practice, especially since extensions to large teams of robots would be even more daunting than they already are in 2D.

One of the areas that promises scalability to larger robot teams and environments is graph-based pursuit-evasion. Also here a number of variants have been considered and the field is vast. The variant most closely related is that of guaranteed graph searching which considers omniscient and arbitrarily fast targets that have to be detected on a vertex or edge of a graph. Again, contamination is often used to represent the possibility that a target is located somewhere in

the graph. An annotated bibliography on guaranteed graph searching is presented in [16]. The goal of guaranteed graph searching is generally to compute a strategy for multiple agents that guarantees target detection while using the least possible number of agents. The variant of graph searching related closest to our paper is known as edge-searching. In edge-searching a graph is initially fully contaminated and can be cleared by moving searchers along its edges. Such moves clear edges and contamination is prevented from spreading through all vertices that are guarded by at least one searcher. An early result regarding the complexity of this problem is given in [34] where it was shown that edge-searching is NP-hard. The question whether re-contamination can improve strategies and hence whether the problem is in NP was addressed in [32] and [4]. Therein it was shown that re-contamination does not matter and one can always find an optimal strategy that avoids re-contamination, also known as a monotone strategy. In [2] the edge-searching problem was generalized to require multiple searchers in order to clear an edge or guard a vertex. Furthermore, the authors consider types of strategies that are contiguous, i.e. strategies for which the cleared edges and vertices of the graph always form a connected sub-graph, also known as connected strategies. The presented algorithm to compute such connected and monotone strategies on trees is, however, not optimal as demonstrated in [30] and [11] due to some flaws in the proofs. In [11] it is also proven that the weighted edge-searching is NP-complete on trees and that there exists a polynomial on tree with bounded degree. It is important to note that connected search is not monotone, i.e. imposing that no re-contamination is allowed when computing connected strategies can lead to strategies that require more agents. An example of this is constructed in [49]. Yet, for practical applications both of these properties, monotonicity and connectedness, are very useful. Monotonicity ensures that areas are only cleared once, which can reduce time and cost while connectedness ensures that there is a safe area through which robots can travel or deploy a communication network.

Applications of graph-based pursuit-evasion algorithms for a robotic application have been discussed in [19, 29, 31]. In this context graphs that have contamination on nodes rather than on edges are used. In the problem from [31], called Graph-Clear, multiple agents may be required to clear vertices and contamination is prevented from spreading by placing robots on edges. So instead of guarding vertices and clearing edges, as in edge-searching, in Graph-Clear edges are guarded and vertices are cleared. This model turns out to be quite different from weighted edge-searching in as much as [11] showed that weighted edge-searching is NP-hard on trees while Graph-Clear can be solved in polynomial time on trees [31]. The work related most closely to our application of graph-searching is [19]. Therein, as in edge-searching, only one agent is required to guard and clear a vertex. Agents can move from one vertex to a neighboring vertex by sliding along an edge and thereby clearing the new vertex. Note that this move does not expose the originating vertex to re-contamination from the new vertex during the sliding move. Due to the similarity of this search to edge-searching the label-based approach from [2] is applied and additional variants with different labeling rules are introduced. Since these algorithms work on trees the GSST algorithm first generates a spanning tree, then computes a strategy on this tree which is then converted back to the original graph. The performance of this approach has been tested in [19] with experiments on several graphs and we adopt the same general approach with regard to generating strategies on the graph. Additionally, in [24] it was shown

that optimal monotone and connected search strategies of a graph form a spanning tree. In other words, the optimal monotone and connected strategy can be found by considering all spanning trees and all valid strategies on these. We shall also briefly refer to this result but in general the enumeration of all spanning trees is only possible for very small environments in the range of less than a dozen nodes. Most environments of interest have many more nodes and in our case around 10^{70} spanning trees. The graph-based approach with randomly sampling multiple spanning trees seems to work well in practice as demonstrated in [19]. Therein a comparison was made on five different graphs between the GSST algorithm and a stochastic hill-climbing approach to compute plans for small teams of agents from [17]. Additional experiments validated the use of random sampling of spanning trees to obtain graph strategies rather than an exhaustive search. Yet, one should note that none of these tests used environments for which the graph-based approach needed more than five agents and all but one needed 3 or less.

For search in real environments these algorithms become useful once a suitable graph can be obtained. In [1] and [28] it was demonstrated that such constructions are feasible and that we can apply graph-based search strategies to coordinate search in real environments. These, however, only consider reducing the number of agents needed for the search and not the time this takes. Time was only considered in [5] and the authors presented 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 road map 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.

Another graph-based related area is known as Moving Target Search first considered in [21]. The assumptions about target motion differs dramatically and targets generally move at bounded speed and their location is known by the searcher. In [27] an A* planner with the graph representing a grid with obstacles is used to solve the problem. Moldenhauer *et al.* [36] presented the *Dynamic Abstract Trail-Max* algorithm for computing strategies in moving target search based on Partial-Refinement A* (PRA*) planning [44]. In contrast to our method, which generates a hierarchical decomposition form elevation maps for computing visibility, PRA* extracts a hierarchy of graphs which is utilized for path planning. Another approach that also assumes that the target's location is known is presented in [3]. Therein the goal is to prevent the evader from escaping from the robot's field of view for as long as possible. The problem is analyzed from a game-theoretic perspective in complex 2D environments. Necessary and sufficient conditions for continued observation and escape are provided, as well as motion strategies that are in Nash equilibrium.

An entirely different graph-based approach is presented in [37]. 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 [1]. 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.

Hierarchical problem decompositions have been used in the past within various other fields, such as quad trees in computer graphics [40] and for efficiently rendering 3D models [15], visual human motion capture [12], in moving target search applied to computer games [36] with known target locations, and path planning [35, 44]. In general, such hierarchical methods work well in practice and are often validated with extensive experiments and demonstrations.

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 agents 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 compared to previous methods based on random sampling, as well as the general feasibility of the approach.

A novel graph structure, embedded in elevation maps, has been presented that is either generated randomly or based on hierarchical sampling and captures visibility information arising in 2.5D problems. Several variants that utilize this information differently have been proposed and evaluated. We have shown empirically that hierarchical sampling combined with sparse edge generation leads to the least amount of needed agents when computing connected 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. This result is likely due to the fact that re-contamination is prohibited. 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 coor-

dination 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 heterogeneous 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.

10 Acknowledgments

This work was supported by AFORS MURI grant FA95500810356 and ONR grant N00014090680.

References

1. M. Lewis A. Kolling, A. Kleiner 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*, pages 4610–4616, 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. S. Bhattacharya and S. Hutchinson. On the existence of nash equilibrium for a visibility based pursuit evasion game. *The International Journal of Robotics Research*, 2009.
4. D. Bienstock and P. Seymour. Monotonicity in graph searching. *Journal of Algorithms*, 12(2):239–245, 1991.
5. 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.
6. J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.
7. F. Bullo, J. Cortés, and S. Martínez. *Distributed control of robotic networks.* Applied Mathematics Series. Princeton University Press, 2009. To appear. Electronically available at http://coordinationbook.info.
8. R.E. Burkard and E. Cela. Linear assignment problems and extensions. Technical report, Karl-Franzens-Univ. Graz & Techn. Univ. Graz, 1998.
9. 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.
10. H. Choset. Coverage for robotics – A survey of recent results. *Annals of Mathematics and Artificial Intelligence*, 31(1-4):113–126, 2001.
11. D. Dereniowski. Connected searching of weighted trees. *Mathematical Foundations of Computer Science 2010*, pages 330–341, 2010.
12. J. Deutscher, A. Davison, and I. Reid. Automatic partitioning of high dimensional search spaces associated with articulated body motion capture. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 2, pages II–669. IEEE, 2001.

13. 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.
14. 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.
15. M. Fan, M. Tang, and J. Dong. A review of real-time terrain rendering techniques. In *Computer Supported Cooperative Work in Design, 2004. Proceedings. The 8th International Conference on*, volume 1, pages 685–691. IEEE, 2003.
16. F. V. Fomin and D. M. Thilikos. An annotated bibliography on guaranteed graph searching. *Theoretical Computer Science*, 399(3):236–245, 2008.
17. B. P. Gerkey, S. Thrun, and G. Gordon. Parallel stochastic hill-climbing with small teams. *Multi-Robot Systems: From Swarms to Intelligent Automata*, 3:65–77, 2005.
18. 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.
19. G. Hollinger, A. Kehagias, and S. Singh. GSST: Anytime guaranteed search. *Autonomous Robots*, 29(1):99–118, 2010.
20. World Bank ImageCat Inc. RIT Haiti earthquake LiDAR. `http://opentopo.sdsc.edu/gridsphere/gridsphere?cid=datasets`, 2010.
21. T. Ishida and R.E. Korf. Moving target search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 204–210. Citeseer, 1991.
22. R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38(4):325–340, 1987.
23. RM Karp. Reducibility among Combinatorial Problems. *Complexity of Computer Computations*, 1972.
24. 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.
25. 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.
26. J.J. Koenderink and A.J. Doorn. The internal representation of solid shape with respect to vision. *Biological cybernetics*, 32(4):211–216, 1979.
27. S. Koenig, M. Likhachev, and X. Sun. Speeding up moving-target search. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–8. ACM, 2007.
28. 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.
29. 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.
30. A. Kolling and S. Carpin. On weighted edge-searching. Technical Report 01, School of Engineering, University of California, Merced, 2009.
31. A. Kolling and S. Carpin. Pursuit-evasion on trees by robot teams. *IEEE Transactions on Robotics*, 26(1):32–47, 2010.
32. A. S. LaPaugh. Recontamination does not help to search a graph. *Journal of the ACM*, 40(2):224–245, 1993.
33. S. Lazebnik. Visibility-based pursuit-evasion in three-dimensional environments. Technical report, University of Illinois at Urbana-Champaign, 2001.
34. N. Megiddo, S. L. Hakimi, M. R. Garey, D. S. Johnson, and C. H. Papadimitriou. The complexity of searching a graph. *Journal of the ACM*, 35(1):18–44, 1988.
35. M. Metea and J. Tsai. Route planning for intelligent autonomous land vehicles using hierarchical terrain representation. In *Robotics and Automation. Proceedings. 1987 IEEE International Conference on*, volume 4, pages 1947 – 1952, mar 1987.
36. Carsten Moldenhauer and Nathan R. Sturtevant. Evaluating strategies for running from the cops. In *Proceedings of the 21st international jont conference on Artifical intelligence*, pages 584–589, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.
37. 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.

38. 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.
39. 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.
40. Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Pub (Sd), 1990.
41. T. Shermer. Recent results in art galleries. *Proceedings of the IEEE*, 80(9):1384–1399, 1992.
42. B. Simov, G. Slutzki, and S. M. LaValle. Clearing a polygon with two 1-searchers. *International Journal of Computational Geometry and Applications*, 19(1):59–92, 2009.
43. B. Steder. Freiburg campus LiDAR data. `http://ais.informatik.uni-freiburg.de/projects/datasets/fr360/`, 2010.
44. Nathan Sturtevant and Michael Buro. Partial pathfinding using map abstraction and refinement. In *Proceedings of the 20th national conference on Artificial intelligence - Volume 3*, pages 1392–1397. AAAI Press, 2005.
45. I. Suzuki and M. Yamashita. Searching for a mobile intruder in a polygonal region. *SIAM Journal on Computing*, 21(5):863–888, 1992.
46. 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.
47. W. T. Tutte. *Graph Theory*. Cambridge University Press, 2001.
48. U.S. Geological Survey (USGS). U.S. Geological Survey (USGS). `http://www.usgs.gov/`, 2010.
49. B. Yang, D. Dyer, and B. Alspach. Sweeping graphs with large clique number. *Lecture notes in computer science*, pages 908–920, 2004.