Multi-agent deployment for visibility coverage in polygonal environments with holes

Karl J. Obermeyer[1*], Anurag Ganguli[2], Francesco Bullo[1]

[1] *Center for Control, Dynamical Systems, and Computation, Univ. of California at Santa Barbara, CA 93106, USA,* karl@engineering.ucsb.edu, bullo@engineering.ucsb.edu
[2] *UtopiaCompression Corporation, 11150 W. Olympic Blvd, Suite 820 Los Angeles, CA 90064, USA* anurag@utopiacompression.com

SUMMARY

This article presents a distributed algorithm for a group of robotic agents with omnidirectional vision to deploy into nonconvex polygonal environments with holes. Agents begin deployment from a common point, possess no prior knowledge of the environment, and operate only under line-of-sight sensing and communication. The objective of the deployment is for the agents to achieve full visibility coverage of the environment while maintaining line-of-sight connectivity with each other. This is achieved by incrementally partitioning the environment into distinct regions, each completely visible from some agent. Proofs are given of (i) convergence, (ii) upper bounds on the time and number of agents required, and (iii) bounds on the memory and communication complexity. Simulation results and description of robust extensions are also included.

KEY WORDS: multi-agent system, sensor network, robotic network, swarm, visibility, line of sight, deployment, coverage

1. INTRODUCTION

Robots are increasingly being used for surveillance missions too dangerous for humans, or which require duty cycles beyond human capacity. In this article we design a distributed algorithm for deploying a group of mobile robotic agents with omnidirectional vision into nonconvex polygonal environments with holes, e.g., an urban or building floor plan. Agents are identical except for their unique identifiers (UIDs), begin deployment from a common point, possess no prior knowledge of the environment, and operate only under line-of-sight sensing and communication. The objective of the deployment is for the agents to achieve full visibility coverage of the environment while maintaining line-of-sight connectivity (at any time the agents' visibility graph consists of a single connected component). We call this the *Distributed Visibility-Based Deployment Problem with Connectivity*. Once deployed, the agents may supply surveillance information to an operator through the ad-hoc line-of-sight communication network. A graphical description of our objective is given in Fig. 1.

Approaches to visibility coverage problems can be divided into two categories: those where the environment is known a priori and those where the environment must be discovered. When the environment is known a priori, a well-known approach is the *Art Gallery Problem* in which one seeks the smallest set of guards such that every point in a polygon is visible to some guard.

*Correspondence to: karl.obermeyer@gmail.com

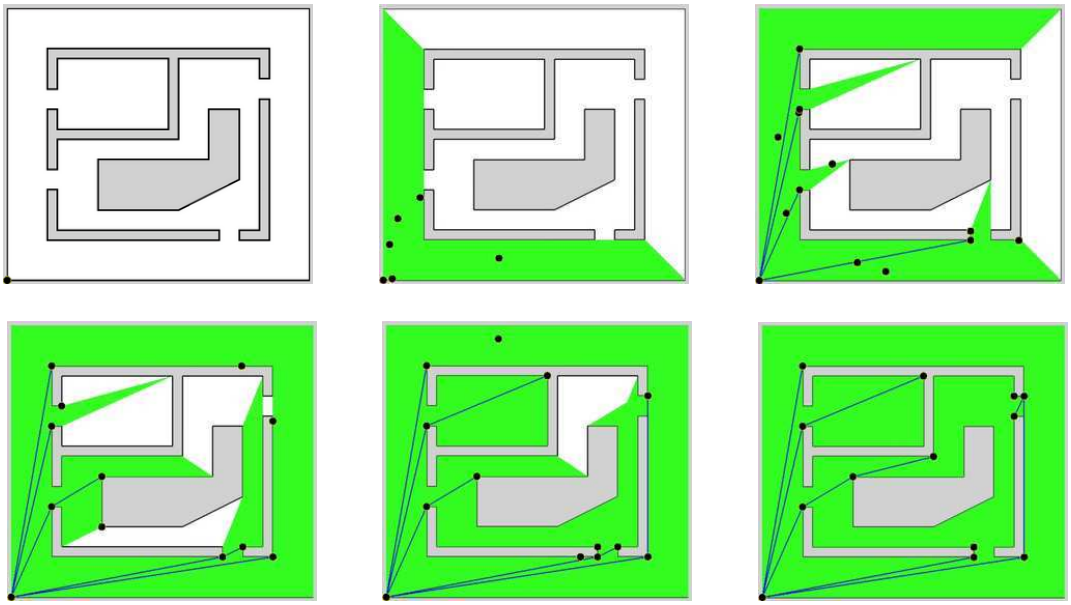


Figure 1. This sequence (left to right, top to bottom) shows a simulation run of the distributed visibility-based deployment algorithm described in Sec. 6. Agents (black disks) initially are colocated in the lower left corner of the environment. As the agents spread out, they claim areas of responsibility (green) which correspond to cells of the incremental partition tree $\mathcal{T}_\mathcal{P}$. Blue lines show line-of-sight connections between agents responsible for neighboring vertices of $\mathcal{T}_\mathcal{P}$. Once agents have settled to their final positions, every point in the environment is visibile to some agent and the agents form a line-of-sight connected network. An animation of this simulation can be viewed at `http://motion.me.ucsb.edu/~karl/movies/dwh.mov`.

This problem has been shown both NP-hard [1] and APX-hard [2] in the number of vertices n representing the environment. The best known approximation algorithms offer solutions only within a factor of $O(\log g)$, where g is the optimum number of agents [3]. The *Art Gallery Problem with Connectivity* is the same as the Art Gallery Problem, but with the additional constraint that the guards' visibility graph must consist of a single connected component, i.e., the guards must form a connected network by line of sight. This problem is also NP-hard in n [4]. Many other variations on the Art Gallery Problem are well surveyed in [5, 6, 7]. The classical *Art Gallery Theorem*, proven first in [8] by induction and in [9] by a beautiful coloring argument, states that $\lfloor \frac{n}{3} \rfloor$ vertex guards[†] are always sufficient and sometimes necessary to cover a polygon with n vertices and no holes. The *Art Gallery Theorem with Holes*, later proven independently by [10] and [11], states that $\lfloor \frac{n+h}{3} \rfloor$ point guards[‡] are always sufficient and sometimes necessary to cover a polygon with n vertices and h holes. If guard connectivity is required, [12] proved by induction and [13] by a coloring argument, that $\lfloor \frac{n-2}{2} \rfloor$ vertex guards are always sufficient and occasionally necessary for polygons without holes. We are not aware of any such bound for connected coverage of polygons with holes. For polygonal environments with holes, centralized camera-placement algorithms described in [14] and [15] take into account practical imaging limitations such as camera range and angle-of-incidence, but at the expense of being able to obtain worst-case bounds as in the Art Gallery Theorems. The constructive proofs of the Art Gallery Theorems rely on global knowledge of the environment and thus are not amenable to emulation by distributed algorithms.

One approach to visibiliy coverage when the environment must be discovered is to first use SLAM (Simultaneous Localization And Mapping) techniques [16] to explore and build a map of the entire

[†] A *vertex guard* is a guard which is located at a vertex of the polygonal environment.
[‡] A *point guard* is a guard which may be located anywhere in the interior or on the boundary of a polygonal environment.

environment, then use a centralized procedure to decide where to send agents. In [17], for example, deployment locations are chosen by a human user after an initial map has been built. Waiting for a complete map of the entire environment to be built before placing agents may not be desirable. In [18] agents fuse sensor data to build only a map of the portion of the environment covered so far, then heuristics are used to deploy agents onto the frontier of the this map, thus repeating this procedure incrementally expands the covered region. For any techniques relying heavily on SLAM, however, synchronization and data fusion can pose significant challenges under communication bandwidth limitations. In [19] agents discover and achieve visibility coverage of an environment not by building a geometric map, but instead by sharing only combinatorial information about the environment, however, the strategy focuses on the theoretical limits of what can be achieved with minimalistic sensing, thus the amount of robot motion required becomes impractical.

Most relevant to and the inspiration for the present work are the distributed visibility-based deployment algorithms, for polygonal environments without holes, developed recently by Ganguli et al [20, 21, 22]. These algorithms are simple, require only limited impact-based communication, and offer worst-case optimal bounds on the number of agents required. The basic strategy is to incrementally construct a so-called *nagivation tree* through the environment. To each vertex in the navigation tree corresponds a region of the the environment which is completely visible from that vertex. As agents move through the environment, they eventually settle on certain nodes of the navigation tree such that the entire environment is covered.

The contribution of this article is the first distributed deployment algorithm which solves, with provable performance, the Distributed Visibility-Based Deployment Problem with Connectivity in polygonal environments with holes. Our algorithm operates using line-of-sight communication and a so-called *partition tree* data structure similar to the *navigation tree* used by Ganguli et al as described above. The algorithms of Ganguli et al fail in polygonal environments with holes because branches of the navigation tree conflict when they wrap around one or more holes. Our algorithm, however, is able to handle such "branch conflicts". Given at least $\lfloor \frac{n+2h-1}{2} \rfloor$ agents in an environment with n vertices and h holes, the deployment is guaranteed to achieve full visibility coverage of the environment in time $\mathcal{O}(n^2 + nh)$, or time $\mathcal{O}(n + h)$ under certain technical conditions. We also prove bounds on the memory and communication complexity. The deployment behaves in simulations as predicted by the theory and can be extended to achieve robustness to agent arrival, agent failure, packet loss, removal of an environment edge (such as an opening door), or deployment from multiple roots.

This article is organized as follows. We begin with some technical definitions in Sec. 2, then a precise statement of the problem and assumptions in Sec. 3. Details on the agents' sensing, dynamics, and communication are given in Sec. 4. Algorithm descriptions, including pseudocode and simulation results, are presented in Sec. 5 and Sec. 6. We conclude in Section 7.

2. NOTATION AND PRELIMINARIES

We begin by introducing some basic notation. The real numbers are represented by $\mathbb{R}$. Given a set, say A, the interior of A is denoted by $\mathrm{int}(A)$, the boundary by ∂A, and the cardinality by $|A|$. Two sets A and B are *openly disjoint* if $\mathrm{int}(A) \cap \mathrm{int}(B) = \emptyset$. Given two points $a, b \in \mathbb{R}^2$, $[a, b]$ is the *closed segment* between a and b. Similarly, $]a, b[$ is the *open segment* between a and b. The number of robotic agents is N and each of these agents has a unique identifier (UID) taking a value in $\{0, \ldots, N-1\}$. Agent positions are $P = (p^{[0]}, \ldots, p^{[N-1]})$, a tuple of points in $\mathbb{R}^2$. Just as $p^{[i]}$ represents the position of agent i, we use such superscripted square brackets with any variable associated with agent i, e.g., as in Table IV.

We turn our attention to the environment, visibility, and graph theoretic concepts. The environment $\mathcal{E}$ is polygonal with vertex set $V_{\mathcal{E}}$, edge set $E_{\mathcal{E}}$, total vertex count $n = |V_{\mathcal{E}}| = |E_{\mathcal{E}}|$, and hole count h. Given any polygon $c \subset \mathcal{E}$, the vertex set of c is V_c and the edge set is E_c. A segment $[a, b]$ is a *diagonal* of $\mathcal{E}$ if (i) a and b are vertices of $\mathcal{E}$, and (ii) $]a, b[\subset \mathrm{int}(\mathcal{E})$. Let e be any point in $\mathcal{E}$. The point e is *visible from* another point $e' \in \mathcal{E}$ if $[e, e'] \subset \mathcal{E}$. The *visibility polygon* $\mathcal{V}(e) \subset \mathcal{E}$ of e is the set of points in $\mathcal{E}$ visible from e (Fig. 2). The *vertex-limited visibility polygon*

$\tilde{\mathcal{V}}(e) \subset \mathcal{V}$ is the visibility polygon $\mathcal{V}(e)$ modified by deleting every vertex which does not coincide with an environment vertex (Fig. 2). A *gap edge* of $\mathcal{V}(e)$ (resp. $\tilde{\mathcal{V}}(e)$) is defined as any line segment $[a, b]$ such that $]a, b[\subset \text{int}(\mathcal{E})$, $[a, b] \subset \partial\mathcal{V}(e)$ (resp. $[a, b] \subset \partial\tilde{\mathcal{V}}(e)$), and it is maximal in the sense that $a, b \in \partial\mathcal{E}$. Note that a gap edge of $\tilde{\mathcal{V}}(e)$ is also a diagonal of $\mathcal{E}$. For short, we refer to the gap edges of $\mathcal{V}(e)$ as the *visibility gaps* of e. A set $R \subset \mathcal{E}$ is *star-convex* if there exists a point $e \in R$ such

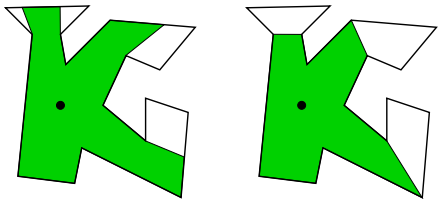


Figure 2. In a simple nonconvex polygonal environment are shown examples of the visibility polygon (green, left) of a point observer (black disk), and the vertex-limited visibility polygon (green, right) of the same point.

that $R \subset \mathcal{V}(e)$. The *kernel* of a star-convex set R, is the set $\{e \in \mathcal{E} | R \subset \mathcal{V}(e)\}$, i.e., all points in R from which all of R is visible. The *visibility graph* $\mathcal{G}_{\text{vis},\mathcal{E}}(P)$ of a set of points P in environment $\mathcal{E}$ is the undirected graph with P as the set of vertices and an edge between two vertices if and only if they are (mutually) visible. A *tree* is a connected graph with no simple cycles. A *rooted tree* is a tree with a special vertex designated as the *root*. The *depth* of a vertex in a rooted tree is the minimum number of edges which must be treversed to reach the root from that vertex. Given a tree $\mathcal{T}$, $V_{\mathcal{T}}$ is its set of vertices and $E_{\mathcal{T}}$ its set of edges.

3. PROBLEM DESCRIPTION AND ASSUMPTIONS

The *Distributed Visibility-Based Deployment Problem with Connectivity* which we solve in the present work is formally stated as follows:

> Design a distributed algorithm for a network of autonomous robotic agents to deploy into an unmapped environment such that from their final positions every point in the environment is visible from some agent. The agents begin deployment from a common point, their visibility graph $\mathcal{G}_{\text{vis},\mathcal{E}}(P)$ is to remain connected, and they are to operate using only information from local sensing and line-of-sight communication.

By local sensing we intend that each agent is able to sense its visibility gaps and relative positions of objects within line of sight. Additionally, we make the following *main assumptions*:

(i) The environment $\mathcal{E}$ is static and consists of a simple polygonal outer boundary together with disjoint simple polygonal holes. By simple we mean that each polygon has a single boundary component, its boundary does not intersect itself, and the number of edges is finite.

(ii) Agents are identical except for their UIDs $(0, \ldots, N-1)$.

(iii) Agents do not obstruct visibility or movement of other agents.

(iv) Agents are able to locally establish a common reference frame.

(v) There are no communication errors nor packet losses.

Later, in Sec. 6.6 we will describe how our nominal deployment algorithm can be extended to relax some assumptions.

4. NETWORK OF VISUALLY-GUIDED AGENTS

In this section we lay down the sensing, dynamic, and communication model for the agents. Each agent has "omnidirectional vision" meaning an agent possesses some device or combination of devices which allows it to sense within line of sight (i) the relative position of another agent, (ii) the relative position of a point on the boundary of the environment, and (iii) the gap edges of its visibility polygon.

For simplicity, we model the agents as point masses with first order dynamics, i.e., agent i may move through $\mathcal{E}$ according to the continuous time control system

$$\dot{p}^{[i]} = u^{[i]}, \tag{1}$$

where the control $u^{[i]}$ is bounded in magnitude by $u_{\max}$. The control action depends on time, values of variables stored in local memory, and the information obtained from communication and sensing. Although we present our algorithms using these first order dynamics, the crucial property for convergence is only that an agent is able to navigate along any (unobstructed) straight line segment between two points in the environment $\mathcal{E}$, thus the deployment algorithm we describe is valid also for higher order dynamics.

The agents' communication graph is precisely their visibility graph $\mathcal{G}_{\mathrm{vis},\mathcal{E}}(P)$, i.e., any *visibility neighbors* (mutually visible agents) may communicate with each other. Agents may send their messages using, e.g., UDP (User Datagram Protocol). Each agent ($i = 0, \ldots, N-1$) stores received messages in a FIFO (First-In-First-Out) buffer In_Buffer$^{[i]}$ until they can be processed. Messages are sent only upon the occurrence of certain asynchronous events and the agents' processors need not be synchronized, thus the agents form an *event-driven asynchronous robotic network* similar to that described, e.g., in [23]. In order for two visibility neighbors to establish a common reference frame, we assume agents are able to solve the *correspondence problem*: the ability to associate the messages they receive with the corresponding robots they can see. This may be accomplished, e.g., by the robots performing localization, however, as mentioned in Sec. 1, this might use up limited communication bandwidth and processing power. Simpler solutions include having agents display different colors, "license plates", or periodic patterns from LEDs [24].

5. INCREMENTAL PARTITION ALGORITHM

We introduce a centralized algorithm to incrementally partition the environment $\mathcal{E}$ into a finite set of openly disjoint star-convex polygonal cells. Roughly, the algorithm operates by choosing at each step a new *vantage point* on the frontier of the uncovered region of the environment, then computing a cell to be covered by that vantage point (each vantage point is in the kernel of its corresponding cell). The frontier is pushed as more and more vantage point - cell pairs are added until eventually the entire environment is covered. The vantage point - cell pairs form a directed rooted tree structure called the *partition tree* $\mathcal{T}_{\mathcal{P}}$. This algorithm is a variation and extension of an incremental partition algorithm used in [22], the main differences being that we have added a protocol for handling holes and adapted the notation to better fit the added complexity of handling holes. The deployment algorithm to be described in Sec. 6 is a distributed emulation of the centralized incremental partition algorithm we present here.

Before examining the precise pseudocode Table I, we informally step through the incremental partition algorithm for the simple example of Fig. 3a-f. This sequence shows the environment partition together with corresponding abstract representations of the partition tree $\mathcal{T}_{\mathcal{P}}$. Each vertex of $\mathcal{T}_{\mathcal{P}}$ is a vantage point - cell pair and edges are based on cell adjacency. Given any vertex of $\mathcal{T}_{\mathcal{P}}$, say (p_ξ, c_ξ), ξ is the *PTVUID (Partition Tree Vertex Unique IDentifier)*. The PTVUID of a vertex at depth d is a d-tuple, e.g., (1), (2,1), or (1,1,1). The symbol $\emptyset$ is used as the root's PTVUID. The algorithm begins with the root vantage point $p_\emptyset$. The cell of $p_\emptyset$ is the grey shaded region $c_\emptyset$ in Fig. 3a, which is the vertex-limited visibility polygon $\tilde{\mathcal{V}}(p_\emptyset)$. According to certain technical criteria, made precise later, child vantage points are chosen on the endpoints of the unexplored gap edges.

Table I. Centralized Incremental Partition Algorithm

```
INCREMENTAL_PARTITION(E, p_∅)
 1: {Compute and Insert Root Vertex into T_P}
 2: c_∅ ← Ṽ(p_∅);
 3: for each gap edge g of c_∅ do
 4:    label g as unexplored in c_∅;
 5: insert (p_∅, c_∅) into T_P;
 6: {Main Loop}
 7: while any cell in T_P has unexplored gap edges do
 8:    c_ζ ← any cell in T_P with unexplored gap edges;
 9:    g ← any unexplored gap edge of c_ζ;
10:    (p_ξ, c_ξ) ← CHILD(E, T_P, ζ, g); {See Tab. II}
11:    {Check for Branch Conflicts}
12:    if there exists any cell c_ξ' in T_P which is in branch conflict with c_ξ then
13:       discard (p_ξ, c_ξ);
14:       label g as phantom_wall in c_ζ;
15:    else
16:       insert (p_ξ, c_ξ) into T_P;
17:       label g as child in c_ζ;
18: return T_P;
```

In Fig. 3a, dashed lines show the unexplored gap edges of $c_\emptyset$. Selecting $p_{(1)}$ as the next vantage point, the corresponding cell $c_{(1)}$ becomes the portion of $\tilde{\mathcal{V}}(p_{(1)})$ which is across the parent gap edge and extends away from the parent's cell. The vantage point $p_{(2)}$ and its cell $c_{(2)}$ are generated in the same way. There are now three vertices, $(p_\emptyset, c_\emptyset)$, $(p_{(1)}, c_{(1)})$, and $(p_{(2)}, c_{(2)})$ in $\mathcal{T}_\mathcal{P}$ (Fig. 3b). In a similar manner, two more vertices, $(p_{(2,1)}, c_{(2,1)})$ and $(p_{(2,1,1)}, c_{(2,1,1)})$, have been added in Fig. 3c. An intersection of positive area is found between cell $c_{(2,1,1)}$ and the cell of another branch of $\mathcal{T}_\mathcal{P}$, namely $c_{(1)}$. To solve this *branch conflict*, the cell $c_{(2,1,1)}$ is discarded and a special marker called a *phantom wall* (thick dashed line in Fig. 3d) is placed where its parent gap edge was. A phantom wall serves to indicate that no branch of $\mathcal{T}_\mathcal{P}$ should cross a particular gap edge. The vertex $(p_{(1,2)}, c_{(1,2)})$ added in Fig. 3e thus can have no children. Finally, Fig. 3f shows the remaining vertices $(p_{(1,1)}, c_{(1,1)})$ and $(p_{(1,1,1)}, c_{(1,1,1)})$ added to $\mathcal{T}_\mathcal{P}$ so that the entire environment is covered and the algorithm terminates.

Now we turn our attention to the pseudocode Table I for a precise description of the algorithm. The input is the environment $\mathcal{E}$ and a single point $p_\emptyset \in V_\mathcal{E}$. The output is the partition tree $\mathcal{T}_\mathcal{P}$. We have seen that each vertex of the partition tree is a vantage point - cell pair. In particular, a cell is a data structure which stores not only a polygonal boundary, but also a label on each of the polygon's gap edges. A gap edge label takes one of four possible values: `parent`, `child`, `unexplored`, or `phantom wall`. These labels allow the following exact definition of the partition tree.

Definition 5.1 (Partition Tree $\mathcal{T}_\mathcal{P}$)
The directed rooted partition tree $\mathcal{T}_\mathcal{P}$ has

(i) vertex set consisting of vantage point - cell pairs produced by the incremental partition algorithm of Table I, and
(ii) a directed edge from vertex (p_ζ, c_ζ) to vertex (p_ξ, c_ξ) if and only if c_ζ has a `child` gap edge which coincides with a `parent` gap edge of c_ξ.

Stepping through the pseudocode Table I, lines 1-5 compute and insert the root vertex $(p_\emptyset, c_\emptyset)$ into $\mathcal{T}_\mathcal{P}$. Upon entering the main loop at line 7, line 8 selects a cell c_ζ arbitrarily from the set of cells in $\mathcal{T}_\mathcal{P}$ which have `unexplored` gap edges. Line 9 selects an arbitrary `unexplored` gap edge g of c_ζ. The next vantage point candidate will be placed on an endpoint of g by a call on line 10 to the CHILD function of Table II. The PTVUID ξ is computed by the successor function on line 1 of Table II. For any d-tuple ζ and positive integer i, $\mathrm{successor}(\zeta, i)$ is simply the $(d+1)$-tuple which is

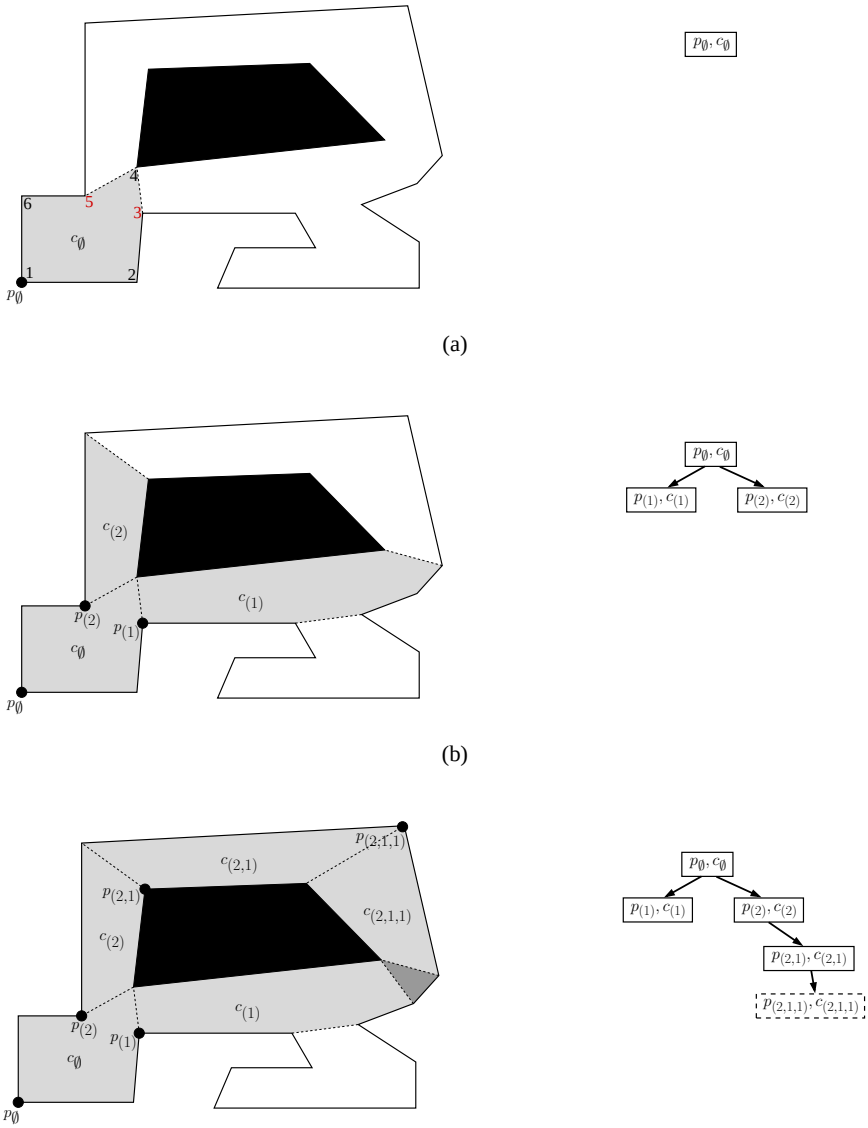
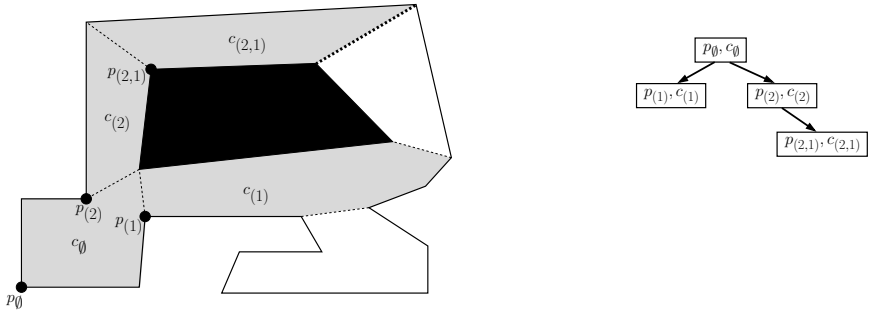
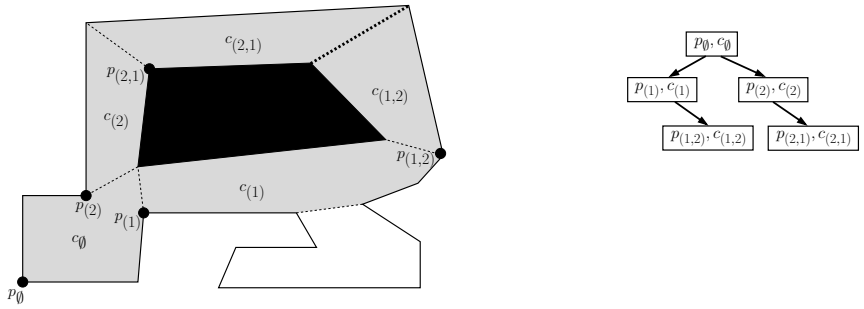


Figure 3. This simple example shows how the incremental partition algorithm of Table I progresses (a)-(f). Cell vantage points are shown by black disks. The portion of the environment $\mathcal{E}$ covered at each stage is shown in grey (left) along with a corresponding abstract depiction of the partition tree (right). A phantom wall (thick dashed line), shown first in (d), comes about when there is a *branch conflict*, i.e., when cells from different branches of the partition tree $\mathcal{T}_\mathcal{P}$ are not openly disjoint. The final partition can be used to triangulate the environment as shown in Fig. 4.

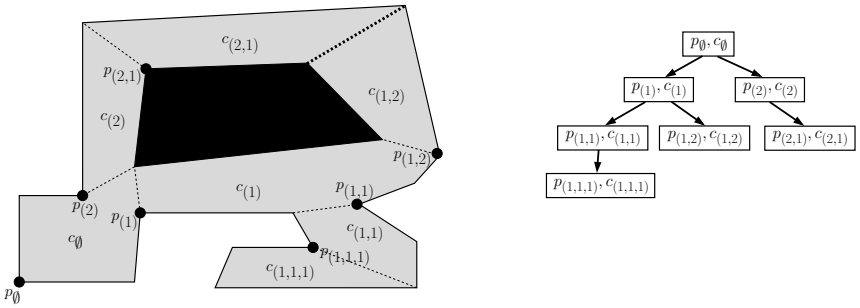
the concatenation of ζ and i, e.g., $\text{successor}((2,1),1)) = (2,1,1)$. The CHILD function constructs a candidate vantage point p_ξ and cell c_ξ as follows. In the typical case, when the parent cell c_ζ has more than three edges, c_ζ's vertices are enumerated counterclockwise from p_ζ, e.g., as $c_\emptyset$'s vertices in Fig. 3a or Fig. 6. In the special case of c_ζ being a triangle, e.g., as the triangular cells in Fig. 6, c_ζ's vertices are enumerated such that the 3 lands on c_ζ's parent gap edge. The vertex of g which is odd in the enumeration is selected as p_ξ. Occasionally there may be *double vantage points* (colocated), e.g., as $p_{(2)}$ and $p_{(3)}$ in Fig. 6. We will see in Sec. 5.1 that this *parity-based vantage point selection*



(d)



(e)



(f)

Figure 3. (continuation)

scheme is important for obtaining a special subset of the vantage points called the *sparse vantage point set*. Returning to Table I, the final portion of the main loop, lines 11-17, checks whether c_ξ is in *branch conflict* or (p_ξ, c_ξ) should be added permanently to $\mathcal{T}_\mathcal{P}$. A cell c_ξ is in branch conflict with another cell $c_{\xi'}$ if and only if c_ξ and $c_{\xi'}$ are not openly disjoint (see Fig. 5). The main algorithm terminates when there are no more unexplored gap edges in $\mathcal{T}_\mathcal{P}$.

An important difference between our incremental partition algorithm and that of Ganguli et al [22] is that the set of cells computed by our incremental partition is not unique. This is because the freedom in choosing cell c_ζ and gap g on lines 8-9 of Table I allows different executions of the algorithm to fill the same part of the environment with different branches of $\mathcal{T}_\mathcal{P}$. This may result in different sets of phantom walls as well. A phantom wall is only created on line 14 of Table I when there is a branch conflict. This discarding may seem computationally wasteful because the

Table II. Incremental Partition Subroutine

CHILD($\mathcal{E}, \mathcal{T}_{\mathcal{P}}, \zeta, g$)

1: $\xi \leftarrow \text{successor}(\zeta, i)$, where g is the ith nonparent gap edge of c_ζ counterclockwise from p_ζ;
2: **if** $|V_{c_\xi}| > 3$ **then**
3: enumerate c_ζ's vertices $1, 2, 3, \ldots$ counterclockwise from p_ζ;
4: **else**
5: enumerate c_ζ's vertices so that p_ζ is assigned 1 and the remaining vertices of c_ζ are assigned 2 and 3
such that the vertex assigned 3 is on the `parent` gap edge of c_ζ;
6: $p_\xi \leftarrow$ vertex on g assigned an odd integer in the enumeration;
7: $c_\xi \leftarrow \tilde{\mathcal{V}}(p_\xi)$;
8: truncate c_ξ at g such that only the portion remains which is across g from p_ζ;
9: delete from c_ξ any vertices which lie across a phantom wall from p_ξ;
10: **for** each gap edge g' of c_ξ **do**
11: **if** $g' == g$ **then**
12: label g' as `parent` in c_ξ;
13: **else if** g' coincides with an existing phantom wall **then**
14: label g' as `phantom_wall` in c_ξ;
15: **else**
16: label g' as `unexplored` in c_ξ;
17: **return** (p_ξ, c_ξ);

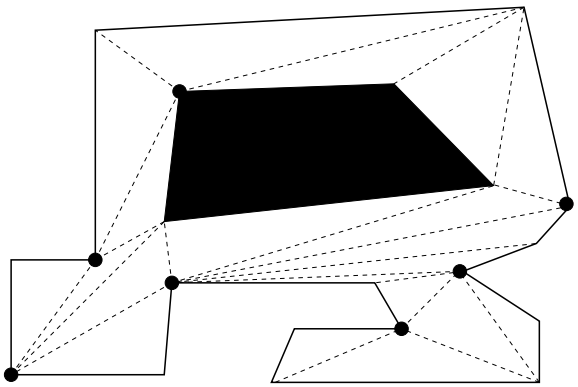


Figure 4. The partition tree produced by the centralized incremental partition algorithm of Table I or the distributed deployment algorithm of Table VI can be used to triangulate an environment, as shown here for the simple example of Fig. 3. The triangulation is constructed by drawing diagonals (dashed lines) from each vantage point (black disks) to the visible environment vertices in its cell.

environment could just be made simply connected by choosing h phantom walls (one for each hole) prior to executing the algorithm. Such an approach, however, would not be amenable to distributed emulation without a priori knowledge of the environment.

The following important properties we prove for the incremental partition algorithm are similar to properties we obtain for the distributed deployment algorithm in Sec. 6.

Lemma 5.2 (Star-Convexity of Partition Cells)
Any partition tree vertex (p_ξ, c_ξ) constructed by the incremental partition algorithm of Table I, has the properties that

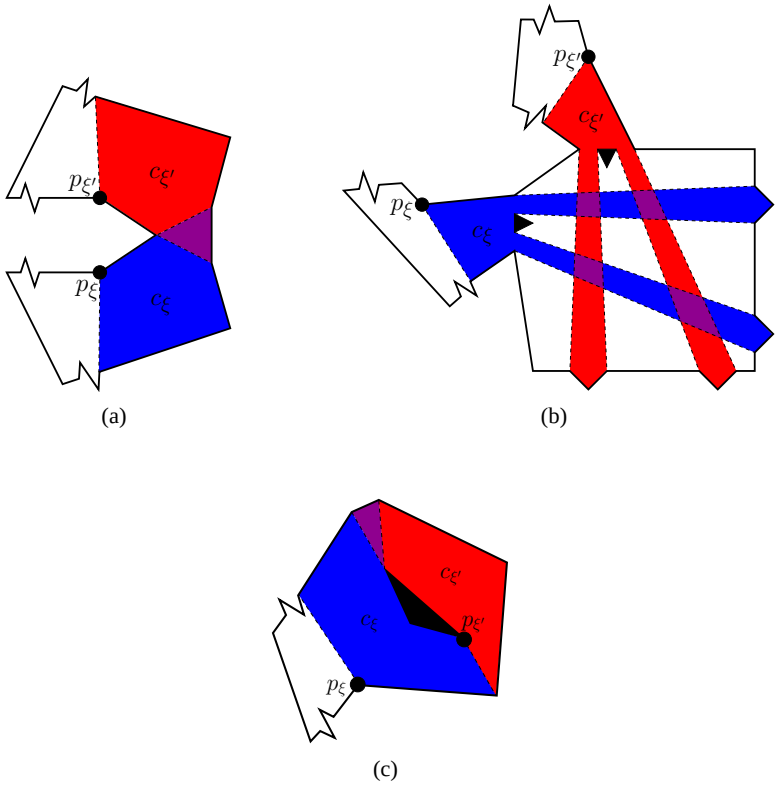


Figure 5. The incremental partition algorithm of Table I and distributed deployment algorithm of Table VI may discard a cell c_ξ if it is in *branch conflict* with another cell $c_{\xi'}$ already in the partition tree, i.e., when c_ξ and $c_{\xi'}$ and are not openly disjoint. In these three examples, blue represents one cell c_ξ, red another cell $c_{\xi'}$, and purple their intersection $c_\xi \cap c_{\xi'}$. A cell can even conflict with it's own parent if they enclose a hole as in (c).

(i) the cell c_ξ is star-convex, and
(ii) the vantage point p_ξ is in the kernel of c_ξ.

Proof
Given a star-convex set, say S, let K be the kernel of S. Suppose that we obtain a new set S' by truncating S at a single line segment l who's endpoints lie on the boundary ∂S. It is easy so see that the kernel of S' contains $K \cap S'$, thus S' must be star-convex if $K \cap S'$ is nonempty. Indeed l could not possibly block line of sight from any point in $K \cap S'$ to any point p in S', otherwise p would have been truncated. Inductively, we can obtain a set S' by truncating the set S at any finite number of line segments and the kernel of S' will be a superset of $S' \cap K$. Now consider a partition tree vertex (p_ξ, c_ξ). By definition, the visibility polygon $\mathcal{V}(p_\xi)$ is star-convex and p_ξ is in the kernel. By the above reasoning, the vertex-limited visibility polygon $\tilde{\mathcal{V}}(p_\xi)$ is also star-convex and has p_ξ in its kernel because $\tilde{\mathcal{V}}(p_\xi)$ can be obtained from $\mathcal{V}(p_\xi)$ by a finite number of line segment truncations (lines 8 and 9 of Table II). Likewise, c_ξ must be star-convex with p_ξ in its kernel because c_ξ is obtained from $\tilde{\mathcal{V}}(p_\xi)$ by a finite number of line segment truncations at the parent gap edge and phantom walls. □

Theorem 5.3 (Properties of the Incremental Partition Algorithm)
Suppose the incremental partition algorithm of Table I is executed on an environment $\mathcal{E}$ with n vertices and h holes. Then

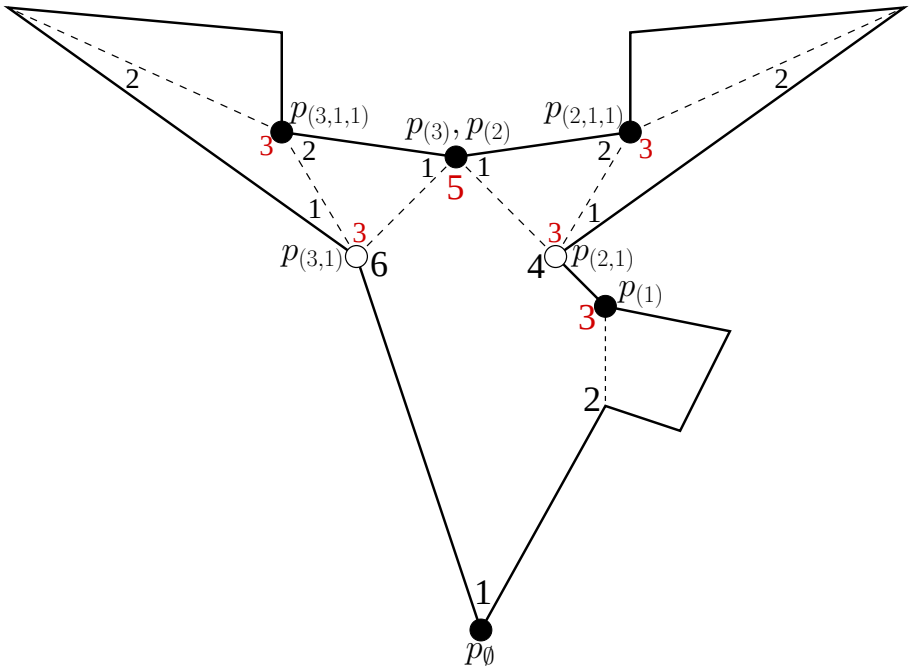


Figure 6. The example used in Fig. 3 showed a typical incremental partition in which there were neither double vantage points nor any triangular cells. This example, on the other hand, shows these special cases. Disks, black or white, show vantage points produced by the incremental partition algorithm of Table I. Integers show enumerations of the cells used for the *parity-based vantage point selection scheme*. The double vantage points $p_{(2)}$ and $p_{(3)}$ are colocated. The cells $c_{(2)}$, $c_{(3)}$, $c_{(2,1)}$, $c_{(3.1)}$, $c_{(2,1,1)}$, and $c_{(3,1,1)}$ are triangular. The vantage points colored black are the *sparse vantage points* found by the postprocessing algorithm of Table III. Under the distributed deployment algorithm of Table VI, robotic agents position themselves at sparse vantage points.

(i) the algorithm returns in finite time a partition tree $\mathcal{T}_\mathcal{P}$ such that every point in the environment is visible to some vantage point,
(ii) the visibility graph of the vantage points $\mathcal{G}_{\text{vis},\mathcal{E}}(\{p_\xi | (p_\xi, c_\xi) \in \mathcal{T}_\mathcal{P}\})$ consists of a single connected component,
(iii) the final number of vertices in $\mathcal{T}_\mathcal{P}$ (and thus the total number of vantage points) is no greater than $n + 2h - 2$,
(iv) there exist environments where the final number of vertices in $\mathcal{T}_\mathcal{P}$ is equal to the upper bound $n + 2h - 2$, and
(v) the final number of phantom walls is precisely h.

Proof
We prove the statements in order. The algorithm processes `unexplored` gap edges one by one and terminates when there are no more `unexplored` gap edges. Once an `unexplored` gap edge has been processed, it is never processed again because its label changes to `phantom_wall` or `child`. Gap edges of cells are diagonals of the environment and there are no more than $\binom{n}{2} = \frac{n^2-n}{2}$ possible diagonals, which is finite, therefore the algorithm must terminate in finite time. Lemma 5.2 guarantees that if the entire environment is covered by cells of $\mathcal{T}_\mathcal{P}$, then every point is visible to some vantage point. Suppose the final set of cells does not cover the entire environment. Then there must be a portion of the environment which is topologically isolated from the rest of the environment by phantom walls, otherwise an `unexplored` gap edge would have expanded into that region. However, this would mean that a phantom wall was created at the `parent` gap edge of a candidate

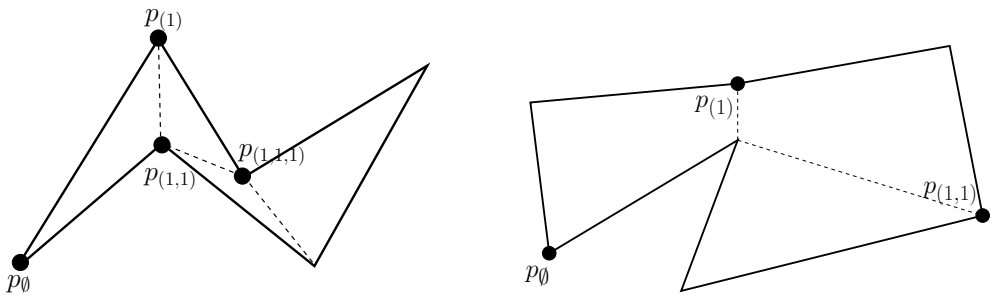


Figure 7. (a) An example of when the final number of vantage points in $\mathcal{T}_\mathcal{P}$ is equal to the upper bound $n + 2h - 2$ given in Theorem 5.3. (b) An example of when the number of points in $\mathbb{R}^2$ where at least one sparse vantage point is located is equal to the upper bound $\left\lfloor \frac{n+2h-1}{2} \right\rfloor$ given in Theorems 5.5 and 6.4.

cell which was not in branch conflict. This is not possible because a phantom wall is only ever created if there is a branch conflict (lines 12-14 Table I). This completes the proof of statement (i).

Statement (ii) follows from Lemma 5.2 together with the fact that every vantage point is placed on the boundary of its parent's cell. Given two vantage points in $\mathcal{T}_\mathcal{P}$, say p_ξ and $p_{\xi'}$, a path through $\mathcal{G}_{\text{vis},\mathcal{E}}(\{p_\xi | (p_\xi, c_\xi) \in \mathcal{T}_\mathcal{P}\})$ from p_ξ to $p_{\xi'}$ can be constructed as follows. Follow parent-child visibility links up to the root vantage point $p_\emptyset$, then follow parent-child visibility links from $p_\emptyset$ down to $p_{\xi'}$. Since such a path can always be constructed between any pair of vantage points, $\mathcal{G}_{\text{vis},\mathcal{E}}(\{p_\xi | (p_\xi, c_\xi) \in \mathcal{T}_\mathcal{P}\})$ must consist of a single connected component.

For statement (iii), we triangulate $\mathcal{E}$ by triangulating the cells of $\mathcal{T}_\mathcal{P}$ individually as in Fig. 4. Each cell c_ξ is triangulated by drawing diagonals from p_ξ to the vertices of c_ξ. The total number of triangles in any triangulation of a polygonal environment with holes is $n + 2h - 2$ (Lemma 5.2 in [6]). Since there is at least one triangle per cell and at most one vantage point per cell, the number of vantage points cannot exceed the maximum number of triangles $n + 2h - 2$.

Statement (iv) is proven by the example in Fig. 7a.

For statement (v), we argue topologically. Suppose the final number of phantom walls were less than h. Then somewhere two branches of the parition tree must share a gap edge with no phantom wall separating them. If this shared gap edge is not a phantom wall, it must be either (1) a child in branch conflict, or (2) unexplored. Either way, the algorithm would have tried to create a cell there but then deleted it and created a phantom wall; a contradiction. Now suppose there were more than h phantom walls. Then a cell would be topologically isolated by phantom walls from the rest of the environment. This is not possible because phantom walls can never be created at the parent-child gap edge between two cells. Since the final number of phantom walls can be neither less nor greater than h, it must be h.

□

5.1. A Sparse Vantage Point Set

Suppose we were to deploy robotic agents onto the vantage points produced by the incremental partition algorithm (one agent per vantage point). Then, as Theorem 5.3 guarantees, we would achieve our goal of complete visibility coverage with connectivity. The number of agents required would be no greater than the number of vantage points, namely $n + 2h - 2$. This upper bound, however, can be greatly improved upon. In order to reduce the number of vantage points agents must deploy to, the postprocessing algorithm in Table III takes the partition tree output by the incremental partition algorithm and labels a subset of the vantage points called the *sparse vantage point set*. Starting at the leaves of the partition tree and working towards the root, vantage points are labeled either `nonsparse` or `sparse` according to criterion on line 2 of Table III. As proven in

Table III. Postprocessing of Partition Tree

LABEL_VANTAGE_POINTS($\mathcal{E}, \mathcal{T}_\mathcal{P}$)
1: **while** there exists a vantage point p_ξ in $\mathcal{T}_\mathcal{P}$ such that p_ξ has not yet been labeled
  and (p_ξ is at a leaf **or** all child vantage points of p_ξ have been labeled) **do**
2: **if** $|V_{c_\xi}| == 3$ **and** p_ξ has exactly one child vantage point labeled `sparse` **then**
3: label p_ξ as `nonsparse`;
4: **else**
5: label p_ξ as `sparse`;

Theorem 5.5 below, the sparse vantage points are suitable for the coverage task and their cardinality has a much better upper bound than the full set of vantage points. All the vantage points in the example of Fig. 3 are sparse. Fig. 6 shows an example of when only a proper subset of the vantage points is sparse.

Lemma 5.4 (Properties of a Child Vantage Point of a Triangular Cell)
Let (p_ξ, c_ξ) be a partition tree vertex constructed by the incremental partition algorithm of Table I and suppose c_ξ has a parent cell c_ζ which is a triangle. Then p_ξ is in the kernel of p_ζ. Furthermore, if p_ζ has a parent vantage point $p_{\zeta'}$ (the grandparent of p_ξ), then p_ξ is visible to $p_{\zeta'}$.

Proof
The kernel of a triangular (and thus convex) cell c_ζ is all of c_ζ. By Lemma 5.2, $p_{\zeta'}$ is in the kernel of $c_{\zeta'}$. According to the parity-based vantage point selection scheme (line 5 of Table II), p_ξ is located at a point common to $c_{\zeta'}$, c_ζ, and c_ξ, therefore p_ξ is in the kernel of c_ζ and visible to $c_{\zeta'}$. □

Theorem 5.5 (Properties of the Sparse Vantage Point Set)
Suppose the incremental partition algorithm of Table I is executed to completion on an environment $\mathcal{E}$ with n vertices and h holes and the vantage points of the resulting partition tree are labeled by the algorithm in Table III. Then

(i) every point in the environment is visible to some sparse vantage point,
(ii) the visibility graph of the sparse vantage points $\mathcal{G}_{\text{vis},\mathcal{E}}(\{p_\xi | (p_\xi, c_\xi) \in \mathcal{T}_\mathcal{P}\})$ consists of a single connected component,
(iii) the number of points in $\mathbb{R}^2$ where at least one sparse vantage point is located is no greater than $\left\lfloor \frac{n+2h-1}{2} \right\rfloor$, and
(iv) there exist environments where the upper bound $\left\lfloor \frac{n+2h-1}{2} \right\rfloor$ in (iii) is met.

Proof

Statements (i) and (ii) follow directly from Lemma 5.4 together with statements (i) and (ii) of Theorem 5.3.

For statement (iii) we use a triangulation argument similar to that used in [22] for environments without holes. We use the same triangulation as in the proof of Theorem 5.3 (Fig. 4). The total number of triangles in any triangulation of a polygonal environment with holes is $n + 2h - 2$ (Lemma 5.2 in [6]). Suppose we can assign at least one unique triangle to $p_\emptyset$ whenever $p_\emptyset$ is sparse and at least two unique triangles to all other sparse vantage point locations. Let N_{sparse} be the number of sparse vantage point locations. Setting $2(N_{\text{sparse}} - 1) + 1 = 2N_{\text{sparse}} - 1$ to be less or equal to the total number of triangles $n + 2h - 2$ and solving for N_{sparse} gives the desired bound

$$N_{\text{sparse}} \leq \left\lfloor \frac{(n + 2h - 2) + 1}{2} \right\rfloor = \left\lfloor \frac{n + 2h - 1}{2} \right\rfloor.$$

Indeed we can make such an assignment of triangles to sparse vantage point locations. Our argument relies on the parity-based vantage point selection scheme and the criterion for labeling a vantage

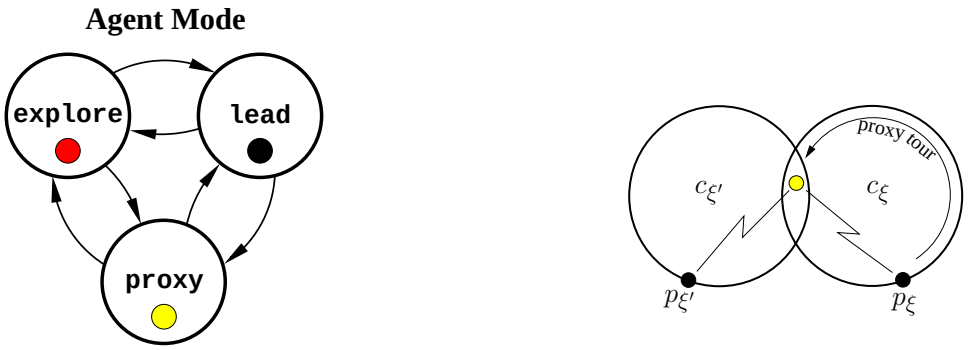


Figure 8. (a) In the distributed deployment algorithm of Table VI, each agent may switch between `lead`, `proxy`, and `explore` mode based on certain asynchronous events. Leader agents are responsible for maintaining a distributed representation of the partition tree $\mathcal{T}_\mathcal{P}$, proxies help establish communication for solving branch conflicts, and explorers systematically navigate through $\mathcal{T}_\mathcal{P}$ in search of opportunities to become a leader or proxy. The agent mode color code is used also in Fig. 10 and 12. (b) Even if a pair of leader agents (black) are not mutually visible, their cells (c_ξ and $c_{\xi'}$) may intersect as in Fig. 5, shown here abstractly by a Venn diagram. Sending a proxy agent (yellow), on a *proxy tour* around one of the cell boundaries guarantees it will enter the cells' intersection so that communication between leaders can be proxied. The leaders can then establish a local common reference frame and compare cell boundaries in order to solve branch conflicts.

point as `sparse` on line 2 of Table III. To any sparse vantage point location, say of p_ξ other than the root, we assign one triangle in the parent cell. The triangle in the parent cell is the triangle formed by its parent gap edge together with its parent's vantage point. To each sparse vantage point location, say of p_ξ, including the root, we assign additionally one triangle in the cell c_ξ. If c_ξ has no children, then any triangle in c_ξ can be assigned to p_ξ. If c_ξ has children (in which case it must have greater than one triangle) we need to check that it has more triangles than child vantage point locations with odd parity. Suppose c_ξ has an even number of edges. Then this number of edges can be written $2m$ where $m \geq 2$. The number of triangles in c_ξ is $2m-2$ and the number of odd parity vertices in c_ξ where child vantage points could be placed is $m-1$. This means at most $m-1$ triangles in c_ξ are assigned to odd parity child vantage point locations, which leaves $(2m-2)-(m-1) = m-1 \geq 1$ triangles to be assigned to the location of p_ξ. The case of c_ξ having an odd number of edges is proven analogously.

Statement (iv) is proven by the example in Fig. 7.

□

6. DISTRIBUTED DEPLOYMENT ALGORITHM

In this section we describe how a group of mobile robotic agents can distributedly emulate the incremental partition and vantage point labeling algorithms of Sec. 5, thus solving the Distributed Visibility-Based Deployment Problem with Connectivity. We first give a rough overview of the algorithm, called DISTRIBUTED_DEPLOYMENT(), and later explain in more detail with aid of the pseudocode in Table VI. Each agent i has a local variable mode$^{[i]}$, among others, which takes a value `lead`, `proxy`, or `explore`. For short, we call an agent in `lead` mode a *leader*, an agent in `proxy` mode a *proxy*, and an agent in `explore` mode an *explorer*. Agents may switch between modes (see Fig. 8a) based on certain asynchronous events. Leaders settle at sparse vantage points and are responsible for maintaining in their memory a distributed representation of the partition tree $\mathcal{T}_\mathcal{P}$ consistent with Definition 5.1. By distributed representation we mean that each leader i retains in its memory up to two *vertices of responsibility*, $(p_1^{[i]}, c_1^{[i]})$ and $(p_2^{[i]}, c_2^{[i]})$, and it knows which gap

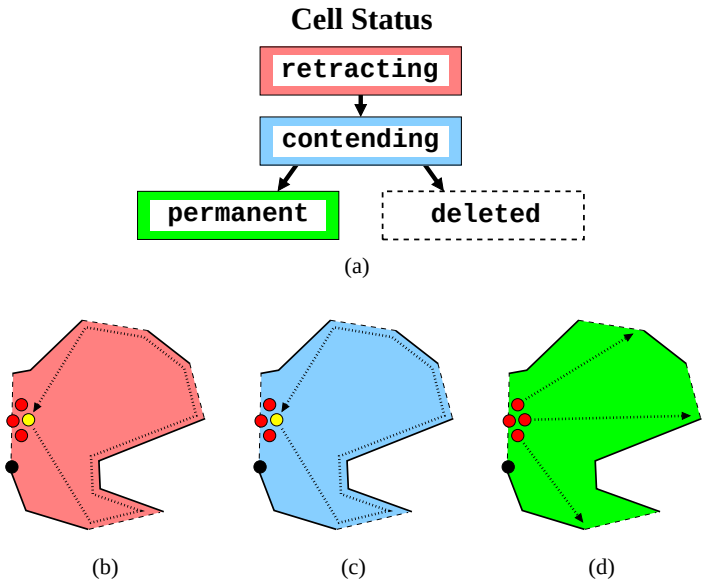


Figure 9. (a) In the distributed deployment algorithm of Table VI, any cell in a leader's memory has a status which takes the value `retracting`, `contending`, or `permanent`. (b) Each cell status is initially `retracting`. The status of a retracting cell is advanced to `contending` after the execution of a proxy tour in which the cell is truncated as necessary to ensure no branch conflict with any permanent cells. (c) In a second proxy tour, a contending cell is deleted if it is found to be in branch conflict with another contending cell of smaller PTVUID (according to total ordering Def. 6.2), otherwise its status is advanced to `permanent`. (d) Only when a cell has attained status `permanent` can any child cells be added at its unexplored gap edges (continued in Fig. 10). The cell status color code is used in Fig. 10 as well as 12.

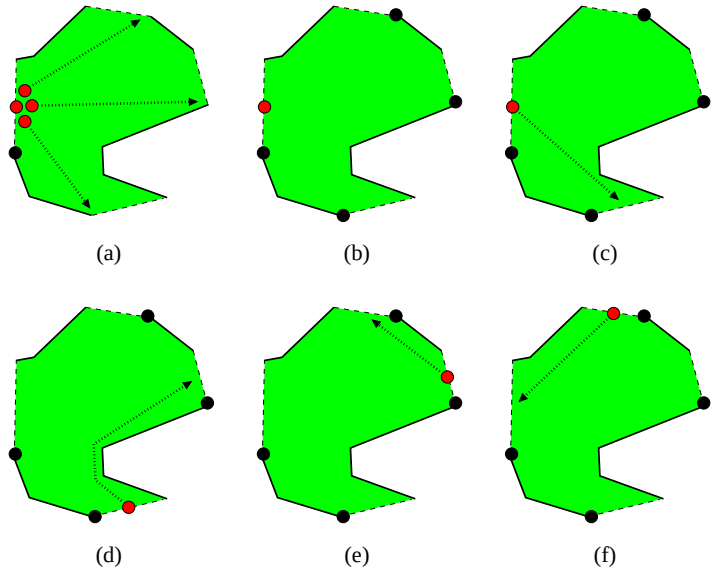


Figure 10. Color codes correspond to those in Fig. 8 and 9. (a,b) Once a cell has status `permanent`, arriving explorer agents can be sent to become leaders at child gap edges. (c-f) Any remaining explorer agents continue systematically navigating the partition tree in search of a leader or proxy tasks they could perform.

Figure 11. In the distributed deployment algorithm of Table VI, explorer agents search the partition tree $\mathcal{T}_\mathcal{P}$ depth-first for leader or proxy tasks they could perform. An agent in a cell, say c_ξ, can always order the gap edges of c_ξ, e.g., counterclockwise from the parent gap edge. The depth-first search progresses by the explorer agent always moving to the next unvisited child or unexplored gap edge in that ordering. The agent thus moves from cell to cell deeper and deeper until a leaf (a vertex with no children) is found. Once at a leaf, the agent backtracks to the most recent vertex with unvisited child or unexplored gap edges and the process continues. As an example, (left) integers (not to be confused with PTVUIDs) show the depth-first order an agent would visit the vertices of $\mathcal{T}_\mathcal{P}$ in Fig. 3f if the gap edges in each cell were ordered couterclockwise from the parent gap edge. If the agent instead uses a gap edge ordering cyclically shifted by one, then (right) shows the different resulting depth-first order. If each agent uses a different gap edge ordering, e.g., cyclically shifted by their UID, then different branches of $\mathcal{T}_\mathcal{P}$ are explored in parallel and the deployment tends to cover the environment more quickly. Cf. Fig. 10.

edges of those vertices lead to the parent and child vertices in $\mathcal{T}_\mathcal{P}$.[§] We call $(p_1^{[i]}, c_1^{[i]})$ the *primary vertex* of agent i and $(p_2^{[i]}, c_2^{[i]})$ the *secondary vertex*. A leader typically has only a primary vertex in its memory and may have also a secondary only if it is either positioned (1) at a double vantage point, or (2) at a sparse vantage point adjacent to a nonsparse vantage point. Each cell in a leader's memory has a status which takes the value `retracting`, `contending`, or `permanent` (see Fig. 9). Only when a cell has attained status `permanent` can any child $\mathcal{T}_\mathcal{P}$ vertices be added at its unexplored gap edges.

Remark 6.1 (3 Cell Statuses)
In our system of three cell statuses, a cell must go through two steps before attaining status `permanent`. Intuitively, the need for two steps arises from the fact that an agent must first determine the boundary of its cell before it can even know what other cells are in branch conflict or place children according to the parity-based vantage point selection scheme. Hence, the first proxy tour allows truncation of the cell boundary at all permanent cells. Only after that, when the boundary is known, is the second proxy tour run and the cell deconflicted with other contending cells. Note that even in the centralized incremental partition algorithm two steps had to be taken by a newly constructed cell: the cell had to be (1) truncated at existing phantom walls, and then (2) deleted if it was in branch conflict.[¶]

The job of a proxy agent is to assist leaders in advancing the status of their cells towards `permanent` by proxying communication with other leaders (see Fig 8b). Any agent which is not a leader or proxy is an explorer. Explorers merely move in depth-first order systematically about $\mathcal{T}_\mathcal{P}$ in search of opportunity to serve as a proxy or leader (see Fig. 10 and 11). To simplify the presentation, let us assume for now that, as in the examples Fig. 3 and Fig. 12, no double vantage points or triangular cells occur. Under this assumption, each leader will be responsible for only one $\mathcal{T}_\mathcal{P}$ vertex, its primary vertex, and all vantage points will be sparse. The deployment begins with all agents colocated at the first vantage point $p_\emptyset$. One agent, say agent 0, is initialized to `lead` mode with the first cell $c_{\xi_1}^{[0]} = c_\emptyset = \tilde{\mathcal{V}}(p_\emptyset)$ in its memory. All other agents are initialized to `explore` mode. Agent 0 can immediately advance the status of $c_\emptyset$ to `permanent` because it cannot possibly

[§]The subscripts of a leader agent's *vertices of responsibility* are not to be confused with PTVUIDs, i.e., $(p_1^{[i]}, c_1^{[i]})$ and $(p_2^{[i]}, c_2^{[i]})$ are not in general the same as $(p_{(1)}, c_{(1)})$ and $(p_{(2)}, c_{(2)})$.
[¶]We did attempt to simplify the distributed deployment alogrithm and make the cells only go through a single step, i.e., a single proxy tour to become permanent, however, there seem to be other difficulties with such an approach, particularly with time complexity bounds.

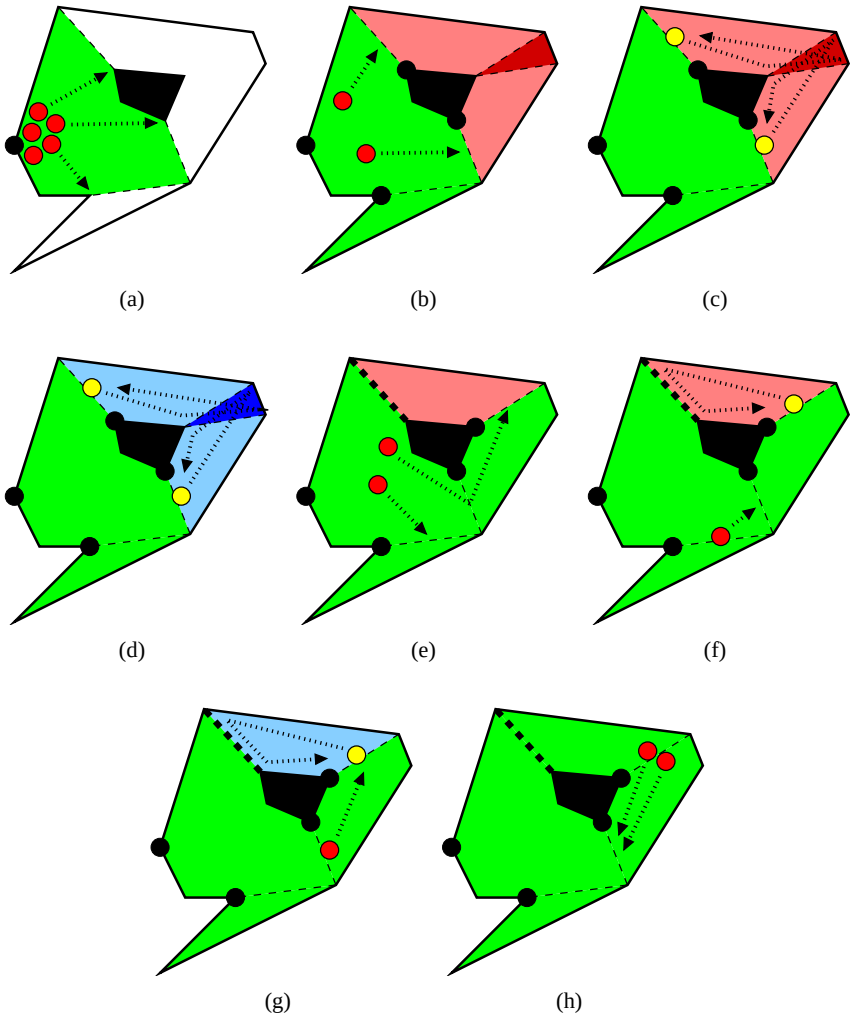


Figure 12. With color codes from Fig. 8 and 9, here is a simple example of agents executing the distributed deployment algorithm of Table VI. (a) Agents enter the environment and the leader initializes the root cell to status `permanent` because no branch conflicts could possibly exist yet. Explorer agents move out to become leaders of child cells. (b) The lower child cell is initialized with status `permanent` because it has no gap edges and thus cannot be in branch conflict. The upper two child cells are initialized to `retracting` because they could be in branch conflict at unexplored gap edges; indeed there is a branch conflict at the dark red overlap region. The remaining explorer agents continue moving out to the new cells. (c) Once the explorers reach the retracting cells, they become proxies and run tours around the cells to check for branch conflict with permanent cells. (d) After the first proxy tours, the child cells' statuses are advanced to `contending` and each proxy run a second tour. (e) During the second proxy tours, the branch conflict is detected between contending cells and the cell with higher PTVUID is deleted. The agents that were in the deleted cell move back up the partition tree and continue exploring depth-first. The other proxy becomes a leader of a new child cell initialized to `retracting`. (f) One of the explorers arrives at the retracting cell and begins a proxy tour to advance the cell to `contending`. (g) The proxy runs a second tour and advances the cell to `permanent` and the partition is completed. (h) Remaining explorers continue navigating the partition tree depth-first in search of tasks; this adds robustness because they will be able to fill in anywhere an agent may fail or a door may open.

be in branch conflict (no other cells even exist yet); in general, however, cells can only transition between statuses when a proxy tour is executed. Agent 0 sees all the explorers in its cell and assigns as many as necessary to become leaders so that there will be one new leader positioned on each

unexplored gap edge of $c_\emptyset$. The new leader agents move concurrently to their new respective vantage points while all remaining explorer agents move towards the next cell in their depth-first ordering. When a leader first arrives at its vantage point, say p_ξ, of the cell c_ξ, it initializes c_ξ to have status `retracting` and boundary equal to the portion of $\tilde{\mathcal{V}}(p_\xi)$ which is across the parent gap edge and extends away from the parent's cell. When an explorer agent comes to such a newly created retracting cell, the leader assigns that explorer to become a proxy and follow a proxy tour which traverses all the gap edges of c_ξ. During the proxy tour, the proxy agent is able to communicate with any leader of a permanent cell that might be in branch conflict with the c_ξ. The cell c_ξ is thus truncated as necessary to ensure it is not in branch conflict with any `permanent` cell. When this first proxy tour is complete, the status of c_ξ is advanced to `contending`. The leader of c_ξ then assigns a second proxy tour which again traverses all the gap edges of c_ξ. During this second proxy tour, the leader communicates, via proxy, with all leaders of contending cells which come into line of sight of the proxy. If a branch conflict is detected between c_ξ and another contending cell, the agents have a *shoot-out*: they compare PTVUIDs of the cells and agree to delete the one which is larger according to the following total ordering.

Definition 6.2 (PTVUID Total Ordering)
Let ξ_1 and ξ_2 be distinct PTVUIDs. If ξ_1 and ξ_2 do not have equal depth, then $\xi_1 < \xi_2$ if and only if the depth of ξ_1 is less than the depth of ξ_2. If ξ_1 and ξ_2 do have equal depth, then $\xi_1 < \xi_2$ if and only if ξ_1 is lexicographically smaller than ξ_2.[‖]

When a cell c_ξ with parent c_ζ is deleted, two things happen: (1) The leader of c_ζ marks a phantom wall at its child gap edge leading to c_ξ, and (2) all agents that were in c_ξ become explorers, move back into c_ζ, and resume depth-first searching for new tasks as in Fig. 12e. If the second proxy tour of a cell c_ξ is completed without c_ξ being deleted, then the status of c_ξ is advanced to `permanent` and its leader may then assign explorers to become leaders of child $\mathcal{T}_\mathcal{P}$ vertices at c_ξ's unexplored gap edges. Agents in different branches of $\mathcal{T}_\mathcal{P}$ create new cells in parallel and run proxy tours in an effort to advance those cells to status `permanent`. New $\mathcal{T}_\mathcal{P}$ vertices can in turn be created at the unexplored gap edges of the new permanent cells and the process continues until, provided there are enough agents, the entire environment is covered and the deployment is complete.

We now turn our attention to pseudocode Table VI to describe DISTRIBUTED_DEPLOYMENT() more precisely. For brevity, this pseudocode is written at a fairly high level. The interested reader may view more implementation details in our technical report available electronically [25]. The algorithm consists of three threads which run concurrently in each agent: communication (lines 1-6), navigation (lines 7-13), and internal state transition (lines 14-21). An outline of the local variables used for these threads is shown in Tables IV and V. The communication thread tracks the internal states of all an agent's visibility neighbors. One could design a custom communication protocol for the deployment which would make more efficient use of communication bandwidth, however, we find it simplifies the presentation to assume agents have direct access to their visibility neighbors' internal states via the data structure Neighbor_Data$^{[i]}$. The navigation thread has the agent follow, at maximum velocity $u_{\max}$, a queue of waypoints called Route$^{[i]}$ as long as the internal state component $c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set is empty (it is only ever nonempty for a proxy agent and its meaning is discussed further in Section 6.2). The waypoints can be represented in a local coordinate system established by the agent every time it enters a new cell, e.g., a polar coordinate system with origin at the cell's vantage point. In the internal state transition thread, an agent switches between `lead`, `proxy`, and `explore` modes. The agent reacts to different asynchronous events depending on what mode it is in. We treat the details of the different mode behaviors in the following Sections 6.1, 6.2, and 6.3.

[‖] For example, $(1) < (2)$ and $(1,3) < (3,2)$, but $(3,2) < (1,3,1)$.

Table IV. Agent Local Variables for Distributed Deployment

Use	Name	Brief Description
Communication	$\text{UID}^{[i]} := i$	agent Unique IDentifier
	$\text{In\_Buffer}^{[i]}$	FIFO queue of messages received from other agents
	$\text{Neighbor\_Data}^{[i]}$	data structure which tracks relevant state information of visibility neighbors
	$\text{state\_change\_interrupt}^{[i]}$	boolean, `true` if and only if internal state has changed between the last and current iteration of the communication thread
	$\text{new\_visible\_agent\_interrupt}^{[i]}$	boolean, `true` if and only if a new agent became visible between the last and current iteration of the communication thread
Navigation	$\text{Route}^{[i]}$	FIFO queue of waypoints
	$p^{[i]}$, $\dot{p}^{[i]}$, u	position, velocity, and velocity input
Internal State	$\text{mode}^{[i]}$	agent mode takes a value `lead`, `proxy`, or `explore`
	$\text{Vantage\_Points}^{[i]} := (p^{[i]}_{\xi_1}, p^{[i]}_{\xi_2})$	vantage points used in `lead` mode for distributed representation of $\mathcal{T}_\mathcal{P}$; may have size 0, 1, or 2; each p_ξ may be labeled either `sparse` or `nonsparse`
	$\text{Cells}^{[i]} := (c^{[i]}_{\xi_1}, c^{[i]}_{\xi_2})$	cells used in `lead` mode for distributed representation of $\mathcal{T}_\mathcal{P}$; may have size 0, 1, or 2; cell fields shown in Tab. V
	$c^{[i]}_{\xi_{\text{proxied}}}$	used in `proxy` mode as local copy of cell being proxied
	$\xi^{[i]}_{\text{current}}$, $\xi^{[i]}_{\text{last}}$	PTVUIDs of current and last $\mathcal{T}_\mathcal{P}$ vertices visited in depth-first search; used in `explore` mode to navigate $\mathcal{T}_\mathcal{P}$

6.1. Leader Behavior

The LEAD() subroutine of the internal state transition thread, called on line 17 of Table VI, is shown in Table VII with the behavior grouped into four sections: attempt cell construction (lines 1-6), assign tasks (lines 7-11), react to deconfliction events (lines 12-20), and propagate sparse vantage point information (lines 21-30). A leader attempts to construct a cell, say c_ξ, whenever it first arrives at p_ξ. In order to guarantee an upper bound on the number of agents required by the deployment (Theorem 6.4), the leader must enforce that any cell it adds to $\mathcal{T}_\mathcal{P}$ contains at least one unique triangle which is not in any other cell of the distributed $\mathcal{T}_\mathcal{P}$ representation. This can be accomplished by the leader first looking at its Neighbor_Data to see if the parent gap edge, call it g, is contained in the cell of any neighbor other than the parent. If not, then the existence of a unique triangle is guaranteed because cell vertices always coincide with environment vertices. In that case the agent safely initializes the cell to `retracting` status and waits for a proxy agent to help it advance the cell's status towards `permanent`. If, however, g is contained in a neighbor cell other

Table V. Cell Data Fields for Distributed Deployment

Name	Brief Description
ξ	PTVUID (Partition Tree Vertex Unique IDentifier)
c_ξ.Boundary	polygonal boundary with each gap edge labeled either as `parent`, `child`, `unexplored`, or `phantom_wall`; child gap edges may be additionally labeled with an agent UID if that agent has been assigned as leader of that gap edge
c_ξ.status	cell status may take a value `retracting`, `contending`, or `permanent`
c_ξ.proxy_uid	UID of agent assigned to proxy c_ξ; takes value $\emptyset$ if no proxy has been assigned
c_ξ.Wait_Set	set of PTVUIDs used by proxy agents to decide when they should wait for another cell's proxy tour to complete before deconfliction can occur, thus preventing race conditions

Table VI. Distributed Deployment Algorithm

DISTRIBUTED_DEPLOYMENT()

```
1:  { Communication Thread }
2:  while true do
3:     in_message ← In_Buffer[i].PopFirst();
4:     update Neighbor_Data[i] according to in_message;
5:     if state_change_interrupt[i] or visible_agent_interrupt[i] then
6:        broadcast internal state information;
7:  { Navigation Thread }
8:  while true do
9:     while Route[i] is not empty and p[i] ≠ Route[i].First() and c[i]_{ξ_proxied}.Wait_Set is empty
       do
10:       u[i] ← velocity with magnitude u_max and direction towards Route[i].First();
11:    u[i] ← 0;
12:    if p[i] == Route[i].First() then
13:       Route[i].PopFirst();
14: { Internal State Transition Thread }
15: while true do
16:    if mode[i] == lead then
17:       LEAD(); { See Tab. VII }
18:    else if mode[i] == proxy then
19:       PROXY(); { See Tab. VIII }
20:    else if mode[i] == explore then
21:       EXPLORE(); { See Tab. IX }
```

than the parent, then the leader may have to either switch to proxy mode to proxy for another leader in line of sight (if the candidate cell is primary), or else wait for the other cell to be proxied (if the candidate cell is secondary). If the agent determines that a contending or permanent cell other than the parent contains g, then it deletes the cell and a phantom wall is labeled.

A leader agent may assign tasks once it has initialized cell(s) in its memory. The assignment may be of an explorer to become a leader of a child vertex, of an explorer to become a proxy, of a leader

Table VII. Distributed Deployment Subroutine

LEAD()

1: { Attempt cell construction }
2: **if** there is a vantage point p_ξ in Vantage_Points$^{[i]}$ for which no cell has yet been constructed **and** $p^{[i]} == p_\xi$ **then**
3: **if** at least one triangle can be made available for c_ξ **then**
4: initialize c_ξ with status `retracting` and insert into Cells$^{[i]}$;
5: **else**
6: delete (p_ξ, c_ξ);
7: { Assign tasks }
8: **if** Cells$^{[i]}$ has a permanent cell with unexplored gap edge g **then**
9: assign an agent to become leader at g;
10: **else if** Cells$^{[i]}$ contains nonpermanent cell $c_\xi^{[i]}$ in need of a proxy **then**
11: assign some agent to proxy $c_\xi^{[i]}$;
12: { React to deconfliction events }
13: **if** cell c_ξ in Cells$^{[i]}$ corresponds to a cell $c^{[j]}_{\xi_{\text{proxied}}}$ in Neighbor_Data$^{[i]}$ **then**
14: update all c_ξ data fields to match $c^{[j]}_{\xi_{\text{proxied}}}$;
15: **if** Neighbor_Data$^{[i]}$ shows a proxy has deleted a cell corresponding to c_ξ in Cells$^{[i]}$ **or** (Neighbor_Data$^{[i]}$ shows contending cell $c^{[j]}_{\xi_{\text{proxied}}}$ in *branch conflict* with contending cell c_ξ in Cells$^{[i]}$ **and** $\xi^{[j]}_{\text{proxied}} < \xi$) **then**
16: delete (p_ξ, c_ξ);
17: **if** Neighbor_Data$^{[i]}$ shows a cell has been deleted at child gap edge g of cell c_ξ in Cells$^{[i]}$ **then**
18: label g as `phantom_wall` in c_ξ;
19: **if** Neighbor_Data$^{[i]}$ shows a proxy tour was successfully completed without deletion for a cell c_ξ in Cells$^{[i]}$ **then**
20: advance c_ξ.status; c_ξ.proxy_uid $\leftarrow \emptyset$;
21: { Propagate sparse vantage point information }
22: **if** there is an unlabeled vantage point p_ξ in Vantage_Points$^{[i]}$ with permanent cell c_ξ in Cells$^{[i]}$ **and** ((p_ξ, c_ξ) is a leaf **or** Cells$^{[i]}$ and Neighbor_Data$^{[i]}$ show all child vantage points have been labeled) **then**
23: **if** $|V_{c_\xi}| == 3$ **and** Cells$^{[i]}$ or Neighbor_Data$^{[i]}$ shows a child vantage point labeled `sparse` **then**
24: label p_ξ as `nonsparse`;
25: **else**
26: label p_ξ as `sparse`;
27: **if** Cells$^{[i]}$ contains exactly one cell c_ξ with p_ξ labeled `sparse` **and** $p^{[i]} == p_\xi$ **and** Neighbor_Data$^{[i]}$ shows a cell c_ζ which is the parent of c_ξ **and** p_ζ is labeled `nonsparse` **then**
28: insert c_ζ into Cells$^{[i]}$ and p_ζ into Vantage_Points$^{[i]}$;
29: **if** Neighbor_Data$^{[i]}$ shows a leader agent j with $p^{[j]}_{\xi_1}$ labeled `sparse` **and** $c^{[i]}_{\xi_2} == c^{[j]}_{\xi_2}$ **and** $\xi^{[j]}_2$ is the parent PTVUID of $\xi^{[i]}_1$ **then**
30: clear $p^{[i]}_{\xi_2}$ and $c^{[i]}_{\xi_2}$; Route$^{[i]} \leftarrow$ straight path to $p^{[i]}_{\xi_1}$;

to become a proxy, of itself to lead a secondary $\mathcal{T}_\mathcal{P}$ vertex which is the child of its primary vertex (this happens when the primary vertex is a triangle), or of another leader to a secondary vertex at a double vantage point. Note that in making the assignments, all vantage points are selected according

Table VIII. Distributed Deployment Subroutine

PROXY()

1: **if** Route$^{[i]}$ is nonempty **and** Neighbor_Data$^{[i]}$ shows proxied cell has not been deleted **then**
2: **if** c_{proxied}.status == `retracting` **then**
3: { Truncate $c_{\xi_{\text{proxied}}}$ at permanent cell }
4: **if** Neighbor_Data$^{[i]}$ shows permanent cell c_ξ in *branch conflict* with $c^{[i]}_{\xi_{\text{proxied}}}$ **then**
5: truncate $c^{[i]}_{\xi_{\text{proxied}}}$ at c_ξ;
6: { Prevent race conditions and deadlock }
7: **if** Neighbor_Data$^{[i]}$ shows contending cell c_ξ in *branch conflict* with $c^{[i]}_{\xi_{\text{proxied}}}$ **and** c_ξ.proxy_uid $\neq \emptyset$ **and** ($\xi^{[i]}_{\text{proxied}} \notin c_\xi$.Wait_Set **or** $\xi < \xi^{[i]}_{\text{proxied}}$) **then**
8: $c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set $\leftarrow c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set $\cup\ \xi$;
9: **else**
10: $c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set $\leftarrow c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set $\setminus\ \xi$;
11: **else if** c_{proxied}.status == `contending` **then**
12: { Shoot-out with other contending cells }
13: **if** (Neighbor_Data$^{[i]}$ shows contending cell c_ξ in *branch conflict* with $c^{[i]}_{\xi_{\text{proxied}}}$ **and** $\xi < \xi^{[i]}_{\text{proxied}}$) **then**
14: delete $c^{[i]}_{\xi_{\text{proxied}}}$; mode$^{[i]} \leftarrow$ `explore`;
15: { Prevent race conditions and deadlock }
16: **if** Neighbor_Data$^{[i]}$ shows retracting cell c_ξ in *branch conflict* with $c^{[i]}_{\xi_{\text{proxied}}}$ **and** c_ξ.proxy_uid $\neq \emptyset$ **and** ($\xi^{[i]}_{\text{proxied}} \notin c_\xi$.Wait_Set **or** $\xi < \xi^{[i]}_{\text{proxied}}$) **then**
17: $c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set $\leftarrow c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set $\cup\ \xi$;
18: **else**
19: $c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set $\leftarrow c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set $\setminus\ \xi$;
20: **else**
21: enter previous mode, explore or lead;

to the same *parity-based vantage point selection scheme* used in the incremental partition algorithm of Sec. 5.

So that the distributed representation of $\mathcal{T}_\mathcal{P}$ remains consistent, a leader must react to several deconfliction events. If a proxy truncates the boundary of a retracting cell, deletes a contending cell, advances the status of a cell, or adds/removes PTVUIDs to a cell's Wait_Set, then the corresponding leader of that cell must do the same. In fact, whenever two agents (either proxies or leaders) communicate and their contending cells are in branch conflict, the cell with lower PTVUID will be deleted. Every such cell deletion results in a phantom wall being marked in the parent cell. Although it is not stated explicitely in the pseudocode, note that when a cell is deleted the leader must wait briefly at the cell's vantage point until any agent that was proxying comes back to the parent cell; otherwise the proxy could lose line of sight with the rest of the network. If a proxy tour is completed successfully without cell deletion, then the cell status is advanced towards `permanent`.

By settling only to sparse vantage points, fewer agents are needed to guarantee full coverage. This is accomplished by agents swaping permanent cells with other leaders in such a way that the information about which vantage points are sparse is propagated up $\mathcal{T}_\mathcal{P}$ whenever a leaf is discovered. Each cell swap involves an acquisition by one agent (lines 27-28) and a corresponding surrender by another (lines 29-30).

Table IX. Distributed Deployment Subroutine

EXPLORE()

1: **if** Neighbor_Data$^{[i]}$ shows a permanent cell c_ξ where $\xi == \xi^{[i]}_{\text{current}}$ **then**
2: $\xi' \leftarrow$ PTVUID of next vertex in *depth-first ordering*;
3: **if** gap edge g at ξ' has already been assigned a leader **then**
4: { Continue exploring }
5: $\xi^{[i]}_{\text{last}} \leftarrow \xi^{[i]}_{\text{current}}$; $\xi^{[i]}_{\text{current}} \leftarrow \xi'$;
6: Route$^{[i]} \leftarrow$ local shortest path to midpoint of g through c_ξ;
7: **else if** gap edge g at ξ' has agent i labeled as its leader **then**
8: { Become leader }
9: mode$^{[i]} \leftarrow$ `lead`; $p^{[i]}_{\xi_1} \leftarrow p_{\xi'}$;
10: Route$^{[i]} \leftarrow$ local shortest path to $p_{\xi'}$ through c_ξ;
11: **else if** Neighbor_Data$^{[i]}$ shows a cell c_ξ such that c_ξ.proxy_uid == i **and** $\xi \neq \xi^{[i]}_{\text{proxied}}$ **then**
12: { Become proxy }
13: mode$^{[i]} \leftarrow$ `proxy`; $c^{[i]}_{\xi_{\text{proxied}}} \leftarrow c_\xi$;
14: Route$^{[i]} \leftarrow$ tour which traverses all gap edges of c_ξ and returns to parent gap edge;
15: **if** Neighbor_Data$^{[i]}$ shows $c_{\xi^{[i]}_{\text{current}}}$ has been deleted **then**
16: { Move up partition tree away from deleted cell }
17: Route$^{[i]} \leftarrow$ local shortest path towards $c_{\xi_{\text{last}}}$; swap $\xi^{[i]}_{\text{last}}$ and $\xi^{[i]}_{\text{current}}$;

6.2. Proxy Behavior

The PROXY() subroutine of the internal state transition thread, called on line 19 of Table VI, is shown in Table VIII. One of two main behaviors are executed depending on whether the proxied cell has status `retracting` (lines 2-10) or `contending` (lines 11-19). Suppose an agent i is proxying for a cell c_ξ in leader agent j's memory. Then agent i keeps a local copy of c_ξ in $c^{[i]}_{\xi_{\text{proxied}}}$ and modifies it during the proxy tour. Agent j updates c_ξ to match $c^{[i]}_{\xi_{\text{proxied}}}$ whenever a change occurs. If agent i is proxying for a retracting cell, then it traverses the gap edges of $c^{[i]}_{\xi_{\text{proxied}}}$ while truncating the cell boundary at any encountered permanent cells in branch conflict. The goal is for the retracting proxied cell to not be in branch conflict with any permanent cells by the end of the proxy tour when its status is advanced to `contending`. If agent i encounters a contending cell, say $c_{\xi'}$, and the criteria on line 7 are satisfied, then agent i must pause its proxy tour, i.e., pause motion until $c_{\xi'}$ becomes permanent or deleted. If the proxy were not to pause, then it would run the risk of the contending cell becoming permanent after the opportunity for the proxy to perform truncation had already passed. The pausing is accomplished by adding ξ' to the cell field $c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set read by the navigation thread. Once the proxy tour is over, the leader of the proxied cell advances the cell's status to `contending` and the proxy agent enters its previous mode, either explore or lead.

If agent i is proxying for a contending cell, then the goal is for that cell to not be in branch conflict with any other contending cells by the end of the proxy tour, if the cell's status is to be advanced to `permanent`. To this end, agent i traverses the gap edges of $c^{[i]}_{\xi_{\text{proxied}}}$ while comparing $\xi^{[i]}_{\text{proxied}}$ with the PTVUID of every encountered contending cell in branch conflict with $c^{[i]}_{\xi_{\text{proxied}}}$. If a contending cell with PTVUID less than $\xi^{[i]}_{\text{proxied}}$ is encountered, then the proxied cell is deleted and agent i heads straight back to the parent gap edge where it will end the proxy tour and enter `explore` mode. If agent i encounters a retracting cell, say $c_{\xi'}$, and the criteria on line 16 are satisfied, then agent i must pause its proxy tour, i.e., pause motion, until $c_{\xi'}$ becomes contending or truncated out of branch conflict. If the proxy were not to pause, then it would run the risk of the retracting

cell becoming contending after the opportunity for the proxy to perform deconfliction had already passed. The pausing is accomplished by adding ξ' to the cell field $c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set read by the navigation thread. Finally, if a contending cell with PTVUID less than $\xi^{[i]}_{\text{proxied}}$ is never encountered, then the leader of the proxied cell advances the cell's status to `permanent` and the proxy agent enters `explore` mode.

Note that the use of PTVUID total ordering (Definition 6.2) on lines 7,13, and 16 of PROXY() precludes the possibility of both (1) *race conditions* in which the status of cells is advanced before the proper branch deconflictions have taken place, and (2) *deadlock* situations where contending and retracting cells are indefinitely waiting for each other.

6.3. Explorer Behavior

The EXPLORE() subroutine of the internal state transition thread, called on line 21 of Table VI, is shown in Table IX. Of all agent modes, `explore` behavior is the simplest because all the agent has to do is navigate $\mathcal{T}_{\mathcal{P}}$ in depth-first order (see Fig. 10 and 11) until a leader agent assigns them to become a leader at an unexplored gap edge or to perform a proxy task. The local shortest paths between cells (lines 6,10, and 17) can be computed quickly and easily by the visibility graph method [26]. If the current cell that an explorer agent is visiting is ever deleted because of branch deconfliction, the explorer simply moves up $\mathcal{T}_{\mathcal{P}}$ and continues depth-first searching. By having each agent use a different gap edge ordering for the depth-first search, the deployment tends to explore many partition tree branches in parallel and thus converge more quickly. In our simulations (Sec. 6.5), we had each agent cyclically shift their gap edge ordering by their UID, subject to the following restriction important for proving an upper bound on number of required agents in Theorem 6.4.

Remark 6.3 (Restriction on Depth-First Orderings)
Each agent in an execution of the distributed deployment may search $\mathcal{T}_{\mathcal{P}}$ depth-first using any child ordering as long as every pair of child vertices adjacent at a double vantage point are visited in the same order by every agent.

6.4. Performance Analysis

The convergence properties of the Distributed Depth-First Connected Deployment Algorithm of Table VI are captured in the following theorems.

Theorem 6.4 (Convergence)
Suppose that N agents are initially colocated at a common point $p_\emptyset \in V_{\mathcal{E}}$ of a polygonal environment $\mathcal{E}$ with n vertices and h holes. If the agents operate according to the Depth-First Connected Deployment Algorithm of Table VI, then

(i) the agents' visibility graph $\mathcal{G}_{\text{vis},\mathcal{E}}(P)$ consists of a single connected component at all times,
(ii) there exists a finite time t^*, such that for all times greater than t^* the set of vertices in the distributed representation of the partition tree $\mathcal{T}_{\mathcal{P}}$ remains fixed,
(iii) if the number of agents $N \geq \lfloor \frac{n+2h-1}{2} \rfloor$, then for all times greater than t^* every point in the environment $\mathcal{E}$ will be visibile to some agent, and there will be no more than h phantom walls, and
(iv) if $N > \lfloor \frac{n+2h-1}{2} \rfloor$, then for all times greater than t^* every cell in the distributed representation of $\mathcal{T}_{\mathcal{P}}$ will have status `permanent` and there will be precisely h phantom walls.

Proof

We prove the statements in order. Nonleader agents, as we have defined their behavior, remain at all times within line of sight of at least one leader agent. Leader agents likewise remain in the kernel of their cell(s) of responsibility and within line of sight of the leader agent responsible for the corresponding parent cell(s). Given any two agents, say i and j, a path can thus be constructed by first following parent-child visibility links from agent i up to the leader agent responsible for

the root, then from the leader agent responsible for the root down to agent j. The agents' visibility graph must therefore consist of a single connected component, which is statement (i).

For statement (ii), we argue similarly to the proof of Theorem 5.3(i). During the deployment, cells are constructed only at unexplored gap edges. A cell either (1) advances though a finite number of status changes or (2) it is deleted during a proxy tour. Either way, each cell is only modified a finite number of times and only one cell is ever created at any particular unexplored gap edge. Since unexplored gap edges are diagonals of the environment and there are only finitely many possible diagonals, we conclude the set of vertices in the distributed representation of $\mathcal{T}_{\mathcal{P}}$ must remain fixed after some finite time t^*.

For statement (iii), we rely on an invariant: during the distributed deployment algorithm, at least two unique triangles can be assigned to every leader agent which has at least one cell of responsibility, other than the root cell, in its memory; at least one unique triangle can be assigned to the leader agent which has the root cell in its memory. One of the triangles is in a leader's own cell (primary or secondary) and its existence is enforced by a leader whenever it initializes a cell in Table VII. The second triangle is in a parent cell of a cell in the agent's memory. The existence of this second triangle is ensured by the depth-first order restriction stipulated in Remark 6.3 together with the parity-based vantage point selection scheme. Remembering that the maximum number of triangles in any triangulation is $n + 2h - 2$ and arguing precisely as we did for the sparse vantage point locations in the proof of Theorem 5.5(iii), we find the number of agents required for full coverage can be no greater than $\lfloor \frac{n+2h-1}{2} \rfloor$. As in the proof of Theorem 5.3(v), the number of phantom walls can be no greater than h because if it where then some cell would be topologically isolated.

Proof of statement (iv) is as for statement (iii), but because there is one extra agent and depth-first is systematic, the extra agent is guaranteed to eventually proxy any remaining nonpermanent cells into `permanent` status and create phantom walls to separate all conflicting partition tree branches. □

Remark 6.5 (Near Optimality without Holes)
As mentioned in Sec. 1, $(n-2)/2$ guards are always sufficient and occasionally necessary for visibility coverage of any polygonal environment without holes. This means that when $h = 0$, the bound on the number of sufficient agents in Theorem 6.4 statement (iii) differs from the worst-case optimal bound by at most one.

Theorem 6.6 (Time to Convergence)
Let $\mathcal{E}$ be an environment as in Theorem 6.4. Assume time for communication and processing are negligible compared with agent travel time and that $\mathcal{E}$ has uniformly bounded diameter as $n \to \infty$. Then the time to convergence t^* in Theorem 6.4 statement (ii) is $\mathcal{O}(n^2 + nh)$. Moreover, if the maximum perimeter length of any vertex-limited visibility polygon in $\mathcal{E}$ is uniformly bounded as $n \to \infty$, then t^* is $\mathcal{O}(n + h)$.

Proof
As in the proof of Theorem 6.4, every cell which is never deleted has at least one unique triangle and there are at most $n + 2h - 2$ triangles total, therefore there are at most $n + 2h - 2$ cells which are never deleted. The maximum number of phantom walls ever created is h (Theorem 6.4). Since cells are only ever deleted when a phantom wall is created, at most h cells are ever deleted. Summing the bounds on the number cells which are and are not deleted, we see the total number of cells any agent must ever visit during the distributed deployment is $n + 2h - 2 + h = n + 3h - 2$. Let l_{d} be the maximum diameter of any vertex-limited visibility polygon in $\mathcal{E}$. Then, neglecting time for proxy tours, an agent executing depth-first search on $\mathcal{T}_{\mathcal{P}}$ will visit every vertex of $\mathcal{T}_{\mathcal{P}}$ in time at most $2u_{\max} l_{\mathrm{d}}(n + 3h - 2)$. Now Let l_{p} be the maximum perimeter length of any vertex-limited visibility polygon in $\mathcal{E}$. Then the total amount of time agents spend on proxy tours, counting two tours for each cell, is $2u_{\max} l_{\mathrm{p}}(n + 3h - 2)$. Exploring and leading agents operate in parallel and at most every agent waits for every proxy tour, so it must be that

$$t^* \leq 2u_{\max}(l_{\mathrm{p}} + l_{\mathrm{d}})(n + 3h - 2).$$

While the diameter of $\mathcal{E}$ being uniformly bounded implies l_{d} is uniform bounded, l_{p} may be $\mathcal{O}(n)$. □

The performance of a distributed algorithm can also be measured by agent memory requirements and the size of messages which must be communicated.

Lemma 6.7 (Memory and Communication Complexity)
Let k be the maximum number of vertices of any vertex-limited visibility polygon in the environment $\mathcal{E}$ and suppose $\mathcal{E}$ is represented with fixed resolution. Then the required memory size for an agent to run the distributed deployment algorithm is $\mathcal{O}(Nk)$ bits and the message size is $\mathcal{O}(k)$ bits.

Proof
The memory required by an agent for its internal state is dominated by its cell(s) of responsibility (of which there are at most two) and proxy cell (at most one). A cell requires $\mathcal{O}(k)$ bits, therefore the internal state requires $\mathcal{O}(k)$ bits. The overall amount of memory in an agent is dominated by Neighbor_Data$^{[i]}$, which holds no more than N internal states, therefore the memory requirement of an agent is $\mathcal{O}(Nk)$. Agents only ever broadcast their internal state, therefore the message size is $\mathcal{O}(k)$. □

6.5. Simulation Results

We used C++ and the VisiLibity library [27] to simulate the Distributed Depth-First Deployment Algorithm of Table VI. An example simulation run is shown in Fig. 1 for an environment with $n = 41$ vertices and $h = 4$ holes. An animation of this simulation can be viewed at `http://motion.me.ucsb.edu/~karl/movies/dwh.mov` . To reduce clutter, we have omitted from this larger example the agent mode and cell status color codes used in Fig. 8, 9, 10, and 12. The environment was fully covered in finite time by only 13 agents, which indeed is less than the upper bound $\lfloor \frac{n+2h-1}{2} \rfloor = 24$ given by Theorem 6.4.

6.6. Extensions

There are several ways that the distributed deployment algorithm can be directly extended for robustness to agent arrival, agent failure, packet loss, and removal of an environment edge. Robustness to agent arrival can be achieved by having any new agents simply enter `explore` mode, setting $\xi^{[i]}_{\mathrm{current}}$ to be the PTVUID of the first cell they land in, and setting $\xi^{[i]}_{\mathrm{last}}$ to be the parent PTVUID of ξ_{current}. The line-of-sight connectivity guaranteed by Theorem 6.4 allows single-agent failures to be detected and handled by having the visibility neighbors of a failed agent move back up the partition tree as necessary to patch the hole left by the failed agent. For robustness to packet loss, agents could add a receipt confirmation and/or parity check protocol. If a portion of the environment were blocked off during the beginning of the deployment but then were revealed by an edge removal (interpreted as the "opening of a door"), the deployment could proceed normally as long as the deleted edge were labeled as an `unexplored` gap edge in the cell it belonged to.

Less trivial extensions include (1) the use of distributed assignment algorithms such as [28, 29] for guiding explorer agents to tasks faster than depth-first search, or (2) performing the deployment from multiple roots, i.e., when different groups of agents begin deployment from different locations. Deployment from multiple roots can be achieved by having the agents tack on a root identifier to their PTVUID, however, it appears this would increase the bound on number of agents required in Theorem 6.4 by up to one agent per root.

7. CONCLUSION

In this article we have presented the first distributed deployment algorithm which solves, with provable performance, the Distributed Visibility-Based Deployment Problem with Connectivity in polygonal environments with holes. We began by designing a centralized incremental partition

algorithm, then obtained the distributed deployment algorithm by asynchronous distributed emulation of the centralized algorithm. Given at least $\lfloor \frac{n+2h-1}{2} \rfloor$ agents in an environment with n vertices and h holes, the deployment is guaranteed to achieve full visibility coverage of the environment in time $\mathcal{O}(n^2 + nh)$, or time $\mathcal{O}(n + h)$ if the maximum perimeter length of any vertex-limited visibility polygon in $\mathcal{E}$ is uniformly bounded as $n \to \infty$. If k is the maximum number of vertices of any vertex-limited visibility polygon in an environment $\mathcal{E}$ represented with fixed resolution, then the required memory size for an agent to run the distributed deployment algorithm is $\mathcal{O}(Nk)$ bits and message size is $\mathcal{O}(k)$ bits. The deployment behaved in simulations as predicted by the theory and can be extended to achieve robustness to agent arrival, agent failure, packet loss, removal of an environment edge (such as an opening door), or deployment from multiple roots.

There are many interesting possibilities for future work in the area of deployment and nonconvex coverage. Among the most prominent are: 3D environments, dynamic environments with moving obstacles, and optimizing different performance measures, e.g., based on continuous instead of binary visibility, or with minimum redundancy requirements.

ACKNOWLEDGEMENT

This work has been supported in part by ONR Award N00014-07-1-0721, NSF Award IIS-0904501, and a DoD SMART fellowship. Thanks to Michael Schuresko (UCSC) and Antonio Franchi (Uni Roma) for helpful comments.

REFERENCES

1. Lee DT, Lin AK. Computational complexity of art gallery problems. *IEEE Transactions on Information Theory* 1986; **32**(2):276–282.
2. Eidenbenz S, Stamm C, Widmayer P. Inapproximability results for guarding polygons and terrains. *Algorithmica* 2001; **31**(1):79–113.
3. Efrat A, Har-Peled S. Guarding galleries and terrains. *Information Processing Letters* 2006; **100**(6):238–245.
4. Liaw BC, Huang NF, Lee RCT. The minimum cooperative guards problem on k-spiral polygons. *Canadian Conference on Computational Geometry*, Waterloo, Canada, 1993; 97–102.
5. Urrutia J. Art gallery and illumination problems. *Handbook of Computational Geometry*, Sack JR, Urrutia J (eds.). North-Holland, 2000; 973–1027.
6. O'Rourke J. *Art Gallery Theorems and Algorithms*. Oxford University Press, 1987.
7. Shermer TC. Recent results in art galleries. *Proceedings of the IEEE* 1992; **80**(9):1384–1399.
8. Chvátal V. A combinatorial theorem in plane geometry. *Journal of Combinatorial Theory. Series B* 1975; **18**:39–41.
9. Fisk S. A short proof of Chvátal's watchman theorem. *Journal of Combinatorial Theory. Series B* 1978; **24**:374.
10. Bjorling-Sachs I, Souvaine D. An efficient algorithm for guard placement in polygons with holes. *Discrete and Computational Geometry* 1995; **13**(1):77–109.
11. Hoffmann F, Kaufmann M, Kriegel K. The art gallery theorem for polygons with holes. *IEEE Symposium on Foundations of Computer Science (FOCS)*, San Juan, Puerto Rico, 1991; 39–48.
12. Hernández-Peñalver G. Controlling guards. *Canadian Conference on Computational Geometry*, Saskatoon, Canada, 1994; 387–392.
13. Pinciu V. A coloring algorithm for finding connected guards in art galleries. *Discrete Mathematical and Theoretical Computer Science, Lecture Notes in Computer Science*, vol. 2731/2003, Springer, 2003; 257–264.
14. González-Baños H, Latombe JC. A randomized art-gallery algorithm for sensor placement. *ACM Symposium on Computational Geometry*, Medford, MA, 2001; 232–240.
15. Erdem UM, Sclaroff S. Automated camera layout to satisfy task-specific and floor plan-specific coverage requirements. *Computer Vision and Image Understanding* 2006; **103**(3):156–169.
16. Thrun S, Burgard W, Fox D. *Probabilistic Robotics*. MIT Press, 2005.
17. Simmons R, Apfelbaum D, Fox D, Goldman R, Haigh K, Musliner D, Pelican M, Thrun S. Coordinated deployment of multiple heterogenous robots. *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, Takamatsu, Japan, 2000; 2254–2260.
18. Howard A, Matarić MJ, Sukhatme GS. An incremental self-deployment algorithm for mobile sensor networks. *Autonomous Robots* 2002; **13**(2):113–126.
19. Suri S, Vicari E, Widmayer P. Simple robots with minimal sensing: From local visibility to global geometry. *International Journal of Robotics Research* 2008; **27**(9):1055–1067.
20. Ganguli A, Cortés J, Bullo F. Distributed deployment of asynchronous guards in art galleries. *American Control Conference*, Minneapolis, MN, 2006; 1416–1421.
21. Ganguli A, Cortés J, Bullo F. Visibility-based multi-agent deployment in orthogonal environments. *American Control Conference*, New York, 2007; 3426–3431.
22. Ganguli A. Motion coordination for mobile robotic networks with visibility sensors. PhD Thesis, Electrical and Computer Engineering Department, University of Illinois at Urbana-Champaign Apr 2007.
23. Bullo F, Cortés J, Martínez S. *Distributed Control of Robotic Networks*. Applied Mathematics Series, Princeton University Press, 2009. Available at http://www.coordinationbook.info.

24. Cruz D, McClintock J, Perteet B, Orqueda OAA, Cao Y, Fierro R. Decentralized cooperative control: A multivehicle platform for research in networked embedded systems. *IEEE Control Systems Magazine* 2007; **27**(3):58–78.
25. Obermeyer K, Ganguli A, Bullo F. Multi-agent deployment for visibility coverage in polygonal environments with holes Aug 2010. Available at `http://arxiv.org/abs/1008.4990`.
26. Nilsson NJ. A mobile automaton: An application of artificial intelligence techniques. *Int. Conference on Artificial Intelligence*, 1969; 509–520.
27. Obermeyer KJ. The VisiLibity library. `http://www.VisiLibity.org` 2008. R-1.
28. Moore BJ, Passino KM. Distributed task assignment for mobile agents. *IEEE Transactions on Automatic Control* 2007; **52**(4):749–753.
29. Zavlanos MM, Spesivtsev L, Pappas GJ. A distributed auction algorithm for the assignment problem. *IEEE Conf. on Decision and Control*, 2008; 1212–1217.