Shortest Paths for Visibility-Based Pursuit-Evasion

Nicholas M. Stiffler Jason M. O'Kane

***Abstract*— We present an algorithm that computes a minimal-cost pursuer trajectory for a single pursuer to solve the visibility-based pursuit-evasion problem in a simply-connected two-dimensional environment. This algorithm improves upon the known algorithm of Guibas, Latombe, LaValle, Lin, and Motwani, which is complete but not optimal. Our algorithm uses a Tour of Segments (ToS) subroutine to construct a pursuer path that minimizes the distance traveled by the pursuer while guaranteeing that all evaders in the environment will be captured. We have implemented our algorithm in simulation and provide results.**

I. INTRODUCTION

The *visibility-based pursuit-evasion problem* requires a pursuer to systematically search an environment to locate one or more evaders, in order to guarantee that all evaders will be found by the pursuer in a finite time. This problem has important applications for intruder detection, search and rescue, and patrolling an area. In all of these applications, the pursuer's success can be measured by the capture time, which measures the worst-case time required to locate the evader. In this paper, we consider an optimal planning algorithm that minimizes the capture time.

The specific problem we consider is a variation on the visibility-based pursuit-evasion problem in which a robot moving through a simply-connected polygonal environment seeks to locate an unknown number of evaders, each of which may move arbitrarily fast. The pursuer has an omni-directional field-of-view that extends to the environment boundary. The goal is to compute a pursuer strategy such that all evaders in the environment lie within the pursuer's field-of-view at some finite time as the pursuer carries out its search strategy, or to identify when no such strategy exists. Guibas, et al. presented a complete algorithm for this problem [3]. However, the authors consider only feasibility and do not attempt to compute optimal strategies. We build upon this work by developing an algorithm that solves the visibility-based pursuit-evasion problem by returning a solution strategy that is optimal in the sense that it minimizes the distance travelled by the pursuer.

We use the same decomposition and pursuit evasion graph as [3], but our algorithm must simultaneously consider multiple paths to each node. Each of these paths is a tour of segments that passes through an ordered sequence of cell boundaries. We introduce a pruning operation to eliminate suboptimal paths, and a forward search algorithm whose termination condition guarantees that an optimal solution will be found.

N. M. Stiffler and J. M. O'Kane are with the Department of Computer Science and Engineering, University of South Carolina, 301 Main St., Columbia, SC 29208, USA. {stifflen,jokane}@cec.sc.edu

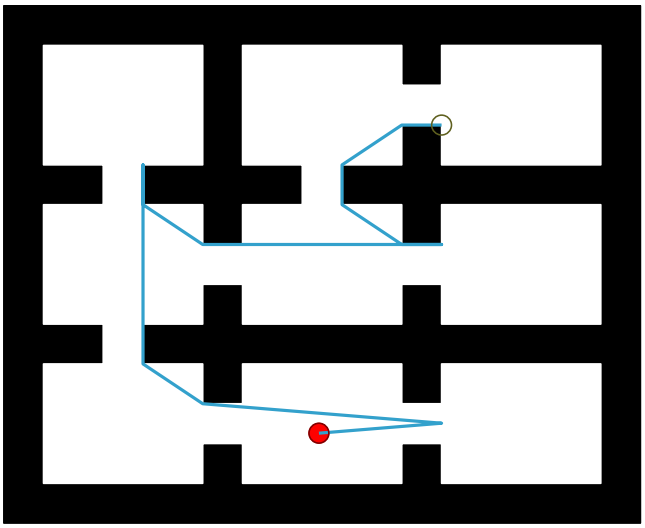


Fig. 1. A pursuer searching an environment. This is the shortest plan from the pursuer's starting position that can guarantee that the pursuer will locate an evader in this environment.

The contribution of this work is a complete algorithm to generate a solution strategy that minimizes the distance traveled by the pursuer. We present simulations that demonstrate that this algorithm succeeds in providing optimal solution strategies.

The remainder of this paper is structured as follows. We begin with a review of related work (Section II), followed by a formal problem statement (Section III), and an overview of the complete algorithm for visibility-based pursuit-evasion (Section IV). This paper then makes several new contributions: We describe an algorithm for computing the shortest path that visits a given sequence of segments in order (Section V), and describe an algorithm that either returns the optimal solution strategy or is able to recognize that no strategy exists (Section VI). Section VII presents our simulations of this algorithm and a quantitative illustration of its effectiveness.

II. RELATED WORK

The pursuit-evasion problem has been studied in a number of contexts. A graph-based pursuit-evasion problem was presented by Parsons [7] in which multiple pursuers and an evader move from vertex to vertex within the graph until a pursuer "captures" an evader by occupying the same vertex as the evader. The visibility-based pursuit-evasion problem introduced by Suzuki and Yamashita [9] is an extension of the watchman route problem [1]. The objective is to compute the shortest path that a guard should take to patrol an entire area populated with obstacles, given only a map of the area.

The capture condition for the general visibility-based pursuit-evasion problem is defined as having an evader lie within the pursuer's capture region. There has been substantial research focused how the visibility-based pursuit-evasion problem changes when a robot has different capture regions. The k-searcher is a pursuer with k visibility beams [6], [9], the ∞-searcher is a pursuer with omni-directional field of view [3], and the ϕ-searcher is a pursuer whose field-of-view is limited to an angle $\phi \in (0, 2\pi]$. Note that all of these approaches consider evaders with unbounded speed.

The case of a velocity-bounded evader was discussed in [10]. An algorithm for visibility-based pursuit-evasion in which the pursuer has a probabilistic model for evader motion was presented in [8]. An algorithm that uses a randomized strategy to solve the visibility-based pursuit-evasion problem appears in [4].

III. PROBLEM STATEMENT

A. Representing the Environment, Pursuer, and Evader

The environment is a simply connected, closed and bounded set $P \subset \mathbb{R}^2$, with polygonal boundary ∂P. For any point $r \in P$, let $VR(r)$ denote the visibility region at point r, which consists of the set of all points in P that are visible from point r. That is, $VR(r)$ contains every point that can be connected to r by a line segment in P. A single pursuer and evader are modeled as points that translate in the polygonal free-space.

Note that a plan that is guaranteed to locate a single evader can also locate multiple evaders, or verify that no evaders exist. Let $\pi(t) \in P$ represent the position of the evader at time $t \geq 0$. The trajectory π is a continuous function $\pi : [0, \infty) \to P$, in which the evader is capable of moving arbitrarily fast (i.e. a finite, unbounded speed) within P. Similarly, let $\gamma(t) \in P$ denote the position of the pursuer at time $t \geq 0$. Without loss of generality, we say that the pursuer moves along γ with maximum speed 1. Figure 2 shows this notation.

The time of capture for an individual evader following trajectory π and a pursuer following trajectory γ is denoted as

$$t_c(\gamma, \pi) = \min\{t \geq 0 \mid \pi(t) \in VR\big(\gamma(t)\big)\}.$$

The pursuer's goal is to capture the evader regardless of the evader's trajectory. A pursuer trajectory γ is a *solution strategy* if there exists a finite time of capture, denoted $t_c(\gamma)$ and defined as

$$t_c(\gamma) = \max_{\pi} \ t_c(\gamma, \pi).$$

The time $t_c(\gamma)$ is the least upper bound for the time of capture over all valid evader trajectories when a pursuer follows trajectory γ . Let γ^* denote a solution strategy that minimizes this capture time:

$$\gamma^* = \operatorname*{argmin}_{\gamma} \ \big(t_c(\gamma)\big).$$

Our goal is to compute this optimal pursuer strategy γ^*.

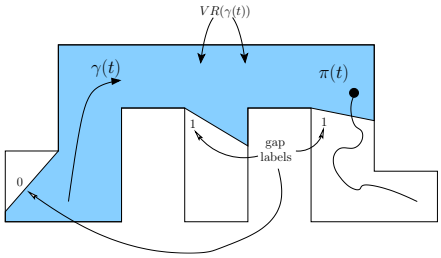


Fig. 2. An illustration of our notation. A single pursuer (triangle) moves through an environment in search of an evader (circle).

IV. COMPLETE SOLUTION FOR VBPE

Before we present our optimal algorithm it is necessary to summarize the work of Guibas et al. [3]. Our algorithm uses similar ideas to find optimal solution strategies for the same problem.

The general idea is that as a pursuer navigates through the environment, it tries to catch the evader in its visibility region, $VR(\gamma(t))$. The edges of the visibility region are either segments along ∂P or edges that cross the interior of P. The edges that cross into the interior of P are referred to as *gap edges*. The pursuer assigns a binary label to each gap. A gap label of 1 means that the region is *contaminated*, which means that it must be searched to ensure that an evader is not hiding beyond the gap. A gap label of 0 means that the region is *cleared*, which means that it is impossible for an evader to be hiding beyond the gap.

Their algorithm begins by decomposing the environment into a collection of convex *conservative regions*, with the property that the gap labels will change only when the pursuer traverses between regions. The decomposition of the environment into conservative regions works by extending rays from inflection points in the environment, and extending rays outwards from pairs of mutually visible environment vertices.The inflection and bitangent ray extensions represent where the pursuer's gap labels change.

There are five events that can cause a change in the pursuer's gap labels.

- An *appear* event signals a gap edge that does not exist in the previous conservative region, is present in the current conservative region. The gap is assigned a cleared (0) label in the current conservative region.
- A *disappear* event signals when a single gap edge that exists in the previous conservative region is no longer present in the current conservative region. The label for this gap is dropped in the current conservative region.
- A *split* event signals when a gap that exists in the previous conservative region is split into multiple gaps in the current conservative region. The new gaps are assigned the value the original gap had in the previous conservative region.
- A *merge* event signals when multiple gaps in the previous conservative region are merged into a single gap in the current conservative region. The new gap is assigned the logical conjunction of the gaps in the

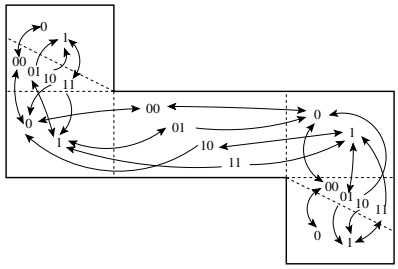


Fig. 3. An example of the Pursuit Evasion Graph for this environment.

previous conservative region.

- A *push* event signals when a gap gets pushed from one environment vertex to another. The gap retains the same label, but is attached to a different environment vertex.

The procedure used in creating the ray extensions provides the following information about what type of event takes place along the boundary of the extension:

(a) Ray extensions caused by an inflection at a single endpoint of an environment edge cause appear and disappear events.
(b) Ray extensions caused by mutually visible environment vertices (where the vertices are not part of the same environment edge) cause split and merge events.
(c) Ray extensions caused by inflections at both endpoints of an environment edge cause push events.

With this information the complete Pursuit-Evasion Graph (PEG) can be constructed as shown in Figure 3. The PEG is a directed graph composed of nodes that contain a gap labeling and a reference to a conservative region, where a node exists for each possible gap label combination for every conservative region. Its edges are the set of critical events that occur from crossing an event boundary from one conservative region to another. The algorithm starts at the PEG-node that contains $\gamma(0)$ with a gap label of $1 \cdots 1$. Using this node as the root of a graph search, the algorithm uses breadth-first search to find a path to a node with a gap label of $0 \cdots 0$. This path through the PEG provides a sequence of conservative regions to visit. The algorithm then constructs a path through P that moves to the centroid of each of those conservative regions in sequence.

V. Optimal Tours of Segments

An important subroutine for our main algorithm will be to compute, given a point p and an ordered collection of segments $(s_1, \ldots, s_n)$, the shortest path that starts at p and visits the segments $(s_1, \ldots, s_n)$ in order. The resulting path is called a *tour of segments* (ToS). Dror, Efrat, Lubiw, and Mitchell showed how to compute such paths in a more general case in which the intermediate steps are polygons rather than segments [2]. We adapt this approach for the specific case of a sequence of segments.

The algorithm proceeds in two basic steps. First, we construct a series of data structures called Shortest Path Maps (SPMs) that allow us to classify the combinatorial structure of shortest paths that visit each segment in the tour. Second, we use a series of point location queries on these SPMs to extract the optimal tour.

A. Step 1: Constructing Shortest Path Maps

The first step of the algorithm is to build an SPM for each segment in the tour. For segment s_1, the resulting SPM is a subdivision of the plane into four regions such that the shortest paths from p to all points in one cell of the subdivision have a combinatorially equivalent shortest path. Figure 4 shows an example. This SPM subdivides the plane at the segment s_1, and at four rays B, C, D, and E, constructed from s_1 and p. The ray B is a ray extension from the left endpoint of s_1 in the direction $\text{left}(s_1) - p$. Likewise, the ray D is a ray extension from the right endpoint of s_1 in the direction $\text{right}(s_1) - p$. Rays C and E are reflections of the rays B and D over the segment s_1.

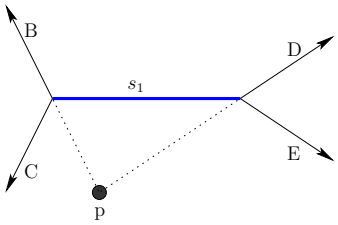


Fig. 4. A single Shortest Path Map. These four rays and one segment subdivide the plane into regions with combinatorially equivalent shortest paths.

Using this structure, and given a query point q, we can compute the shortest path from p to q via s_1, as shown in Figure 5. There are four general cases.

(a) If q is between rays B and C, then the shortest path from p to q via s_1 is a "left turn" at the left endpoint of s_1.
(b) If q is between rays D and E, then the shortest path from p to q via s_1 is a "right turn" at the right endpoint of s_1.
(c) If q is on or above s_1 and between rays B and D, then the shortest path from p to q via s_1 is to go "through" s_1 directly to q.
(d) If q is beneath s_1 and between rays C and E, then the shortest path from p to q via s_1 is to "bounce" off of s_1.

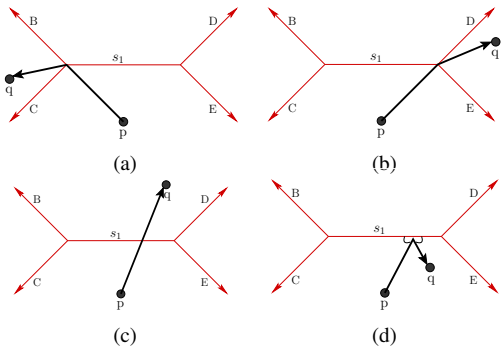


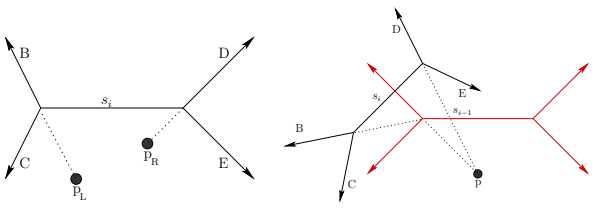
Fig. 5. The SPM for the first segment s_1 divides the plane according to the combinatorial structure of the shortest path from p to s_1 to a query point q.

To compute the SPM for segment s_i with $i > 1$, we need a more general structure with two start points p_L and p_R which are determined by point location queries in the previous SPMs, as shown in Figure 6a. The construction is similar to the construction at s_1 as described above, except that rays B and C are constructed using p_L, whereas rays D and E are constructed using p_R.

Algorithm 1 shows the process for selecting these two start points. Throughout we use a point-location subroutine called LOCATE that takes as input the index of a specific SPM and a query point q, and returns the region containing q in that SPM. The idea is to recurse backward through the previously constructed SPMs until we reach a left or right turn. The intuition is that these left and right turns are points that are known with certainty to lie on the ToS; in contrast, for through or bounce steps, additional segments may change that portion of the ToS. Figure 6b illustrates this process.

Algorithm 1 SELECTSTARTPOINT(Index i, Point q)

```
if i = 0 then
    return p
end if
r ← LOCATE(i − 1, q)
if r = LEFT then
    return left(s_{i−1})
else if r = RIGHT then
    return right(s_{i−1})
else if r = THROUGH then
    return SELECTSTARTPOINT(i − 1, q)
else if r = BOUNCE then
    return SELECTSTARTPOINT(i−1, REFLECT(q, s_{i−1}))
end if
```



(a) (b)

Fig. 6. Computing the SPM for segment s_i depends on the SPM for segment s_{i-1}.

B. Step 2: Extracting the Optimal Tour of Segments

The final step of our ToS algorithm is to extract the complete optimal tour using the SPMs described above. The algorithm begins by computing the set of intersection points between s_n and all of the SPMs. This produces a subdivision of s_n into a collection of $O(n)$ subsegments. Note that, due to our construction of the subsegments, each subsegment is fully contained in a single region of each SPM. For each subsegment we locate the largest i for which the subsegment is in either the left or right region of the SPM for s_i. Then we construct the complete path by executing EXTRACTPATH$(i-1, \text{left}(s_i))$ or EXTRACTPATH$(i-1, \text{right}(s_i))$ respectively, appended with the shortest direct path from that point to the subsegment, with appropriate reflections for bounce regions along the way from s_i to the subsegment of s_n. If there is no such i, the technique is similar, but uses the start point p instead, treating it as a degenerate segment. Pseudocode for the path extraction for each candidate can be found in Algorithm 2; the intuition is to traverse backward through the SPMs to p, adding a new edge to the path at each left, right, and bounce event. In this way, each subsegment generates a candidate path, and the ToS algorithm simply selects the shortest from among these candidate paths.

Algorithm 2 EXTRACTPATH(Index i, Point q)

```
if i = 0 then
    return p
end if
r ← LOCATE(i, q)
if r = LEFT then
    return (EXTRACTPATH(i − 1, left(s_i)), left(s_i))
else if r = RIGHT then
    return (EXTRACTPATH(i − 1, right(s_i)), right(s_i))
else if r = THROUGH then
    return EXTRACTPATH(i − 1, q)
else if r = BOUNCE then
    r ← REFLECT(q, s_i)          reflect point across segment
    P ← EXTRACTPATH(i − 1, r)
    return (P, LINEINTERSECTION(s_i, (r, P.back())))
end if
```

VI. ALGORITHM DESCRIPTION

This section introduces our algorithm for optimal VBPE. We begin with a cell decomposition of the environment into conservative regions to compute the entire PEG (Section VI-A). Starting from the initial PEG-node, we maintain a priority queue of PEG-nodes waiting to be expanded. At each such node N, we maintain a collection of segment sequences that the pursuer can cross to reach node N from the root node. As the search discovers new segment sequences, it performs a pruning operation based on a provable criterion for dominance of one sequence over another. Section VI-B describes this pruning process. The priority queue orders the nodes according the length of the shortest ToS across all of the segment sequences stored at each node. Using this priority queue, the algorithm performs a forward search to construct a complete solution strategy. Section VI-C presents the details of this forward search.

A. Cell Decomposition and Pursuit-Evasion Graph

We use the technique of Guibas, et al. [3], described in Section IV, to perform a cell decomposition of P into conservative regions. Atop this decomposition, we compute the PEG, which has one node for each unique sequence of gap labels at each conservative region.

B. Path Dominance and Pruning

In this section, we characterize solution strategies in terms of the sequences of conservative region boundary edges they cross, provide a dominance relation that allows our algorithm to discard many suboptimal partial boundary edge sequences, and present an algorithm that performs this pruning.

First, we can make a connection between the concept of a solution strategy for the pursuer and the sequence of conservative region edges crossed by the pursuer while executing that strategy.

Theorem 1: Let γ denote a solution strategy, and let $(s_1, \ldots, s_n)$ denote the sequence of conservative region boundary edges crossed by γ. Then any other pursuer trajectory γ' that crosses $(s_1, \ldots, s_n)$ in the same order is also a solution strategy.

Proof: Notice that γ and γ' must traverse same sequence of conservative regions. But because those regions are conservative, the gap labels achieved by γ and γ' remain identical at each conservative region in the sequence. Therefore γ' reaches, as does γ, a PEG-node whose gap labels are all 0, and γ' is a solution strategy. ■

Because of the connection between solution strategies and segment sequences established by Theorem 1, our algorithm maintains, for each PEG-node N, a collection of segment sequences known to reach N.

Note the pursuer can only follow such a segment sequence if each successive pair of segments lies on a single conservative region. Specifically, we require that for any sequence $(s_1, \ldots, s_n)$ stored at a PEG-node N, we have that s_i and s_{i+1} lie on the same conservative region, for each $i = 1, \ldots, n-1$, and s_n lies in the conservative region corresponding to the node N. We call such a sequence a *valid sequence* for N. If there exists a solution strategy that passes through a valid sequence, we call this sequence a *solution sequence*.

Notice, however, that there are potentially infinitely many valid segment sequences for any PEG-node. To counteract this problem, we introduce a notion of *dominance* between segment sequences, which enables our search algorithm to prune many suboptimal sequences.

Definition 1: A segment sequence $\hat{s} = (s_1, \ldots, s_n)$ is *dominated* by a segment sequence $\hat{r} = (r_1, \ldots, r_m)$ if:

(a) The tours of segments from the start point p through $\hat{s}$, and from the start point p through $\hat{r}$ both terminate in the same conservative region.

(b) For any segment sequence $\hat{a} = a_1, \ldots, a_k$ for which $(\hat{s}, \hat{a})$ and $(\hat{r}, \hat{a})$ are valid sequences, we have
$$\ell(ToS(\hat{s}, \hat{a})) \geq \ell(ToS(\hat{r}, \hat{a})) \ ,$$
in which ℓ denotes the length of a path.

(c) Every gap labeled 0 in the PEG-node reached by $\hat{s}$ is also labeled 0 in the PEG-node reached by $\hat{r}$.

A sufficient condition for stating that parts (a) and (b) of Definition 1 are satisfied for segment sequences $r_1, \ldots, r_m$ and $s_1, \ldots, s_n$ is to show that
$$\ell(ToS(\hat{r}, s_n, \hat{a})) \leq \ell(ToS(\hat{s}, \hat{a})).$$
Note however, that the addition of future segments $\hat{a}$ can change the combinatorial structure of the prefix portion of either of these ToSes. These changes can occur when the complete ToS passes through one of the endpoints of s_n, one of the intersection points of s_n and the SPM for s_{n-1}, or one of the intersection points of s_n and the SPM for r_m. Because the structure for both ToS prefixes is stable between these test points, it is sufficient to compare the ToS lengths only at these points. For this process we make a general position assumption that the segments s_n and r_m are distinct. If this is not the case, we instead consider the intersection points between s_n and the SPM for r_{m-1}.

The following observation establishes a connection between this dominance relation and optimal solution sequences.

Observation 1: Let $\hat{s} = (s_1, \ldots, s_n)$, $\hat{a} = (a_1, \ldots, a_k)$, $\hat{r} = (r_1, \ldots, r_m)$, such that $(\hat{s}, \hat{a})$ and $(\hat{r}, \hat{a})$ are valid solution sequences. If $\hat{r}$ dominates $\hat{s}$, then $(\hat{s}, \hat{a})$ is not the optimal solution strategy.

As a result of this observation, our algorithm prunes any dominated sequence that is generated in the course of this search. The pruning operation is called when a new sequence is generated during node expansion. For a sequence to be added, it must not be dominated by any sequence belonging to a PEG-node that satisfies part (c) of Definition 1.

C. The Forward Search

Our planner to compute an optimal solution strategy uses a forward search approach [5]. We maintain a priority queue of PEG-nodes to expand, ordered by the length of the shortest ToS stored at each node. At each iteration, we expand one node N by extending each of the segment sequences associated with it, by appending each edge of the conservative region containing N. If those segment sequences are not dominated, they are inserted into the appropriate nodes.

The termination conditions for our algorithm are twofold. First, if the priority queue becomes empty, the search terminates. Second, if the length of the shortest ToS for the head node in our priority-queue is greater than the length of best solution strategy seen so far, no additional node expansions will generate a shorter solution strategy, so the search terminates successfully. If a solution strategy has not been found by the conclusion of the search, the algorithm reports that no solution strategy exists. Otherwise, the algorithm returns the optimal solution strategy found during the forward search.

We know that the returned solution is an optimal solution strategy because the termination condition for the search guarantees that no solution sequence with a ToS shorter than our solution exists, and the ToS algorithm guarantees that we have found the shortest path that visits that segment sequence.

VII. RESULTS

In this section, we provide simulated results for our algorithm compared to the complete algorithm presented by Guibas et al., shown in Figure 7. We ran our simulations on three separate environments:

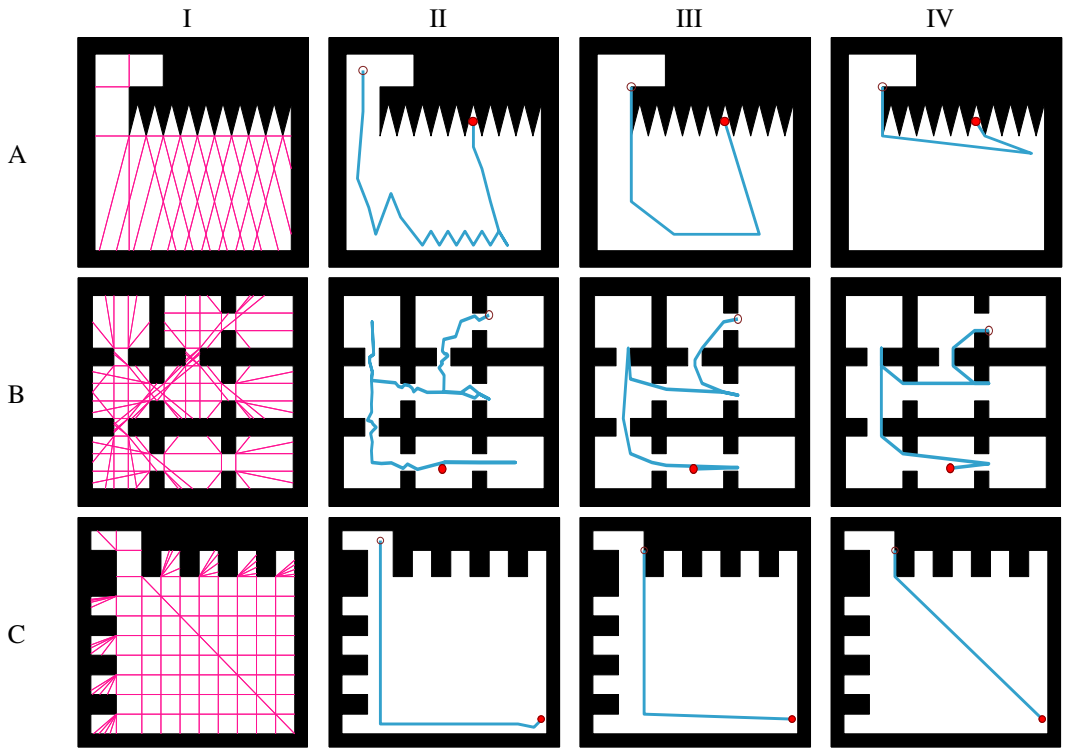


Fig. 7. Column I shows the cell decomposition for three environments. Column II shows the raw solution strategy generated by the Guibas et al. algorithm. Column III shows the shortest solution strategy that visits the same sequence of conservative regions as the Guibas et al. algorithm. Column IV shows the optimal solution strategy generated by our algorithm.

1) The environment in Figure 7A have 57 conservative regions, with a total of $21,806$ PEG-nodes. The number of gaps per conservative region is at most 11.
2) The environment in Figure 7B have 213 conservative regions, with a total of $26,620$ PEG-nodes. The number of gaps per conservative region is at most 11.
3) The environment in Figure 7C have 125 conservative regions, with a total of $35,530$ PEG-nodes. The number of gaps per conservative region is at most 10.

The below table shows the results of these simulations. Note that while the algorithm presented by Guibas et al. minimizes the number of PEG-nodes visited in a solution strategy, this is not a sufficient condition for generating optimal solution strategies. For comparison we applied a post processing step to the Guibas paths to find the shortest pursuer trajectory that visits the same sequence of conservative regions. These results clearly illustrate that the optimal solution strategy is not necessarily a solution strategy that visits the fewest PEG-nodes.

Environment	Guibas et al. path length (raw)	Guibas et al. path length (ToS)	Stiffler-O'Kane path length
1. Fig 7A	34.3705	22.3838	15.7671
2. Fig 7B	47.9996	34.1259	31.3484
3. Fig 7C	26.8714	24.066	17.8706

ACKNOWLEDGMENT

We acknowledge support for this work from NSF (IIS-0953503).

REFERENCES

[1] W. Chin and S. Ntafos. Shortest watchman routes in simple polygons. *Discrete Computational Geometry*, 6(1):9–31, 1991.
[2] M. Dror, A. Efrat, A. Lubiw, and J. Mitchell. Touring a sequence of polygons. In *Proc. ACM Symposium on Theory of Computing*, pages 473–482. ACM Press, 2003.
[3] L. J. Guibas, J.-C. Latombe, S. M. LaValle, D. Lin, and R. Motwani. Visibility-based pursuit-evasion in a polygonal environment. *International Journal of Computational Geometry and Applications*, 9(5):471–494, 1999.
[4] V. Isler, S. Kannan, and S. Khanna. Randomized pursuit-evasion in a polygonal environment. *IEEE Transactions on Robotics*, 5(21):864–875, 2005.
[5] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Also available at http://planning.cs.uiuc.edu/.
[6] S. M. LaValle, B. Simov, and G. Slutzki. An algorithm for searching a polygonal region with a flashlight. *International Journal of Computational Geometry and Applications*, 12(1-2):87–113, 2002.
[7] T.D. Parsons. Pursuit-evasion in a graph. In Y. Alani and D. R. Lick, editors, *Theory and Application of Graphs*, pages 426–41, Springer-Verlag, Berlin, 1976.
[8] N. M. Stiffler and J. M. O'Kane. Visibility-based pursuit-evasion with probabilistic evader models. In *Proc. IEEE International Conference on Robotics and Automation*, 2011.
[9] I. Suzuki and M. Yamashita. Searching for a mobile intruder in a polygonal region. *SIAM Journal on Computing*, 21(5):863–888, 1992.
[10] B. Tovar and S. M. LaValle. Visibility-based pursuit-evasion with bounded speed. In *Proc. Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2006.