A Sampling-Based Algorithm for Multi-Robot Visibility-Based Pursuit-Evasion

Nicholas M. Stiffler Jason M. O'Kane

***Abstract*— We introduce a probabilistically complete algorithm for solving a visibility-based pursuit-evasion problem in two-dimensional polygonal environments with multiple pursuers. The inputs for our algorithm are an environment and the initial positions of the pursuers. The output is a joint strategy for the pursuers that guarantees that the evader has been captured. We create a Sample-Generated Pursuit-Evasion Graph (SG-PEG) that utilizes an abstract sample generator to search the pursuers' joint configuration space for a pursuer solution strategy that captures the evaders. We implemented our algorithm in simulation and provide results.**

I. Introduction

There are many variants of the *pursuit-evasion* problem. The common theme amongst them is that one group of agents, the "pursuers", attempts to track members of another group, the "evaders".

This paper considers a specific variant of the pursuit-evasion problem called *visibility-based pursuit-evasion*, which requires the pursuer(s) to systematically search an environment to locate the evaders, ensuring that all evaders will be found by the pursuers in a finite time. The specific problem we consider is a visibility-based pursuit-evasion problem that utilizes a team of pursuers. The pursuers move through a polygonal environment seeking to locate an unknown number of evaders, which move at a finite but unbounded speed. The pursuers have an omni-directional field-of-view that extends to the environment boundary. The goal is to find a joint strategy for the pursuers that ensures that all of the evaders are seen.

The visibility-based pursuit-evasion problem has an extra layer of complexity beyond the standard motion planning problem because of its capture guarantee. It is not enough to simply select a standard motion planner and attempt to generate a path for each pursuer through the environment. To guarantee that the pursuer strategy does indeed capture an evader if one exists, the planner must also reason about the regions of the environment that are not currently in the pursuers' visual field-of-view and how these regions interact with one another as the pursuers move within the environment.

Two dominant threads of research involve the number of deployable pursuers available to solve the visibility-based pursuit-evasion problem. Using only a single pursuer, there are results that yield complete [4], randomized [8], and optimal [22] solutions, as well as many other variants discussed in Section II. A consequence of using only a single pursuer is that these algorithms are only applicable when the environment can be represented as a simply-connected polygon.

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}@cse.sc.edu

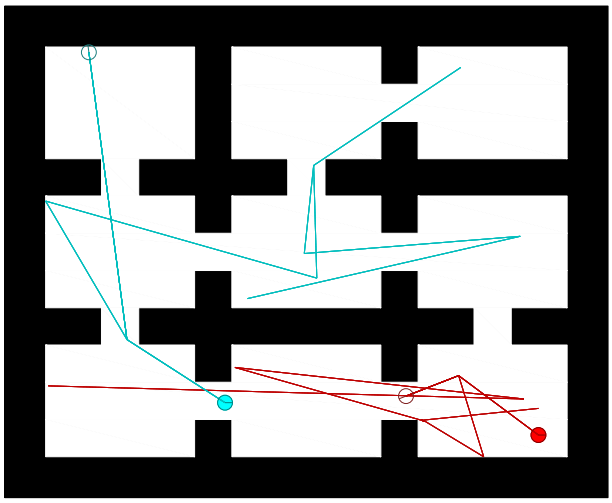


Fig. 1: A pursuer strategy generated by our algorithm. Filled circles represent the pursuers' initial positions and open circles represent their goal positions.

The authors considered the multiple pursuer visibility-based pursuit-evasion problem [23] in the past. In that work, we introduced a centralized algorithm for computing a pursuer solution strategy. The general idea is to create a Cylindrical Algebraic Decomposition (CAD) of the pursuers' joint configuration space by using polynomials that capture where critical changes to the regions of the environment hidden from the pursuers occur. Then we compute the adjacency graph for the CAD and construct a Pursuit Evasion Graph (PEG) induced by the adjacency graph. A search through the PEG can produce one of the following outcomes: the search can reach a vertex where the pursuers' motions up to this point ensure that the evader has been captured, or the search terminates without finding a solution and produces a statement recognizing that no solution exists. The drawback of the technique is the computational complexity required to construct the CAD and perform the adjacency test, which is doubly exponential in the number of pursuers. This paper differs from that work in that we no longer discretize the configuration space and maintain a CAD nor compute the adjacency graph.

The main contribution of this work is a probabilistically complete algorithm for multiple pursuer visibility-based pursuit-evasion that generates a solution strategy for the

pursuers to execute (Figure 1) through the joint configuration space. Our algorithm creates a graph that maintains the pursuers' information state, and utilizes a sample generator that we treat as a "black box" to reason about unexplored areas in the pursuers' joint configuration space. Our algorithm has some similarity to the Probabilistic Roadmap (PRM) algorithm [10], but differs in that our algorithm maintains information concerning the areas of the environment where the evader might be. The need for this additional information complicates both the update operations for the graph and the selection of samples.

The remainder of this paper is structured as follows. In Section II we discuss related work to our problem. Section III contains a formal problem statement. A formal definition for the area not visible to the pursuers, called *shadows*, appears in Section IV. This paper makes several new contributions:

1) We introduce a graph that maintains a representation of the reachable parts of the pursuers' joint information space and provide details about its construction (Section V).
2) We introduce an algorithm that uses this graph to search for a pursuer solution strategy (Section VI).
3) We present simulation results (Section VII) that show our algorithm's ability to generate solution strategies for various sample generators.

Discussion and concluding remarks appear in Section VIII.

II. Related Work

The pursuit-evasion problem was originally posed in the context of differential games [5], [7]. The lion and man game and the homicidal chauffeur are two such differential games. In the lion and man game, a lion tries to capture a man who is trying to escape [15], [21]. In game theory, the homicidal chauffeur is a pursuit-evasion problem which pits a slowly moving but highly maneuverable runner against the driver of a vehicle, which is faster but less maneuverable, who is attempting to run him over [7], [19].

The first recognized instance of pursuit-evasion on a graph is the Parsons problem [17]. The idea behind the Parsons problem, also known as the edge-searching problem, is to determine a sequence of moves for the pursuers that can detect all intruders in a graph using the least number of pursuers. A move consists of either placing or removing a pursuer on a vertex, or sliding it along an edge. A vertex is considered guarded as long as it has at least one pursuer on it, and any evader located therein or attempting to pass through will be detected. A sliding move detects any evader on an edge.

The visibility-based pursuit-evasion problem was proposed by Suzuki and Yamashita [24] as a geometric formulation of the graph-based problem and can be viewed as an extension of the watchman route problem [1], in which 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.

A. Single Pursuer

The capture condition for the general visibility-based pursuit-evasion problem [4] is defined as having an evader lie within a 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 [14], [24], the ∞-searcher is a pursuer with omnidirectional field of view [4], [16], and the ϕ-searcher is a pursuer whose field-of-view [3] is limited to an angle $\phi \in (0, 2\pi]$. Note that all of these approaches consider evaders with unbounded speed.

Others have studied scenarios where there are additional constraints, such as the case of curved environments [13], an unknown environment [20], a maximum bounded speed for the pursuer [26], or constraints similar to those of a typical bug algorithm [18].

B. Multiple Pursuer

As a result of the problem complexity, there is a wide range of literature with differing techniques attempting to solve the multi-robot visibility-based pursuit-evasion problem. One technique organizes the pursuers into teams, whose joint sensing capability are a set of moving lines, each of which is spanned between obstacles. By using these teams of robots as sweep lines, the authors guarantee detection of the evaders [12]. Other researchers have used a mixed integer linear programming approach to solve a multi-pursuer visibility-based pursuit-evasion problem [25]. Another approach involves maintaining complete coverage of the frontier [2]. There are other variants of the pursuit-evasion problem where the pursuers are teams of unmanned aerial vehicles [11].

III. Problem Statement

Portions of this section appear in the authors' prior work [23] and are included here for completeness.

A. Representing the environment, evaders, and pursuers

1) The environment: The environment is a polygonal free space, defined as a closed and bounded set $F \subset \mathbb{R}^2$, with a polygonal boundary ∂F. The environment is composed of m vertices.

2) The evader: The evader is modeled as a point in F that can translate within the environment. Let $e(t) \in F$ denote the position of the evader at time $t \geq 0$. The path e is a continuous function $e : [0, \infty) \to F$, in which the evader is capable of moving arbitrarily fast (i.e. a finite, unbounded speed) within F. Note that, by assuming that there is a single evader, we have not sacrificed any generality. If the pursuers can guarantee the capture of a single evader, then the same strategy can locate multiple evaders, or confirm that no evaders exist.

3) The pursuers: A collection of n identical pursuers cooperatively move to locate the evader. We assume that the pursuers know F, and that they are centrally coordinated. Therefore, from a given collection of starting positions, the pursuers' motions can be described by a continuous function

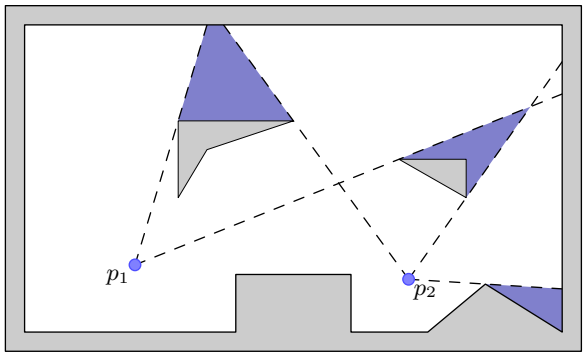


Fig. 2: An environment with two pursuers and three shadows.

$p : [0, \infty) \to F^n$, so that $p(t) \in F^n$ denotes the joint configuration of the pursuers at time $t \geq 0$. The function p, which our algorithm generates, is called a *joint motion strategy* for the pursuers. We use the notation $p^i(t) \in F$ to refer to the position of pursuer i at time t. Likewise, $x^i(t)$ and $y^i(t)$ denote the horizontal and vertical coordinates of $p^i(t)$. Without loss of generality, we assume that the pursuers move with maximum speed 1.

Each pursuer carries a sensor that can detect the evader. The sensor is omnidirectional and has unlimited range, but cannot see through obstacles. For any point $q \in F$, let $V(q)$ denote the visibility region at point q, which consists of the set of all points in F that are visible from point q. That is, $V(q)$ contains every point that can be connected to q by a line segment in F. Note that $V(q)$ is a closed set.

B. Capture conditions

The pursuers' goal is to guarantee the capture of the evader for any continuous evader trajectory.

Definition *A joint motion strategy is a* solution strategy *if, for any continuous evader trajectory $e : [0, \infty) \to F$, there exists some time t and some pursuer i such that $e(t) \in V\big(p^i(t)\big)$.*

IV. Shadows

The key difficulty in locating our evader is that the pursuers can not, in general, see the entire environment at once. This section contains some definitions for describing and reasoning about the portion of the environment that is not visible to the pursuers at any particular time.

Definition *The portion of the environment not visible to the pursuers at time t is called the shadow region $S(t)$, and defined as*

$$S(t) = F - \bigcup_{i=1,\ldots,n} V\big(p_i(t)\big).$$

Note that the shadow region may contain zero or more nonempty path-connected components, as seen in Figure 2.

Definition *A* shadow *is a maximal path connected component of the shadow region.*

Notice that $S(t)$ is the union of the shadows at time t. The important idea is that the evader, if it has not been captured, is always contained in exactly one shadow, in which it can move freely.

As the pursuers move, the shadows can change in any of four ways, called *shadow events*.

- *Appear*: A new shadow can appear, when a previously visible part of the environment becomes hidden.
- *Disappear*: An existing shadow can disappear, when one or more pursuers move to locations from which that region is visible.
- *Split*: A shadow can split into multiple shadows, when the pursuers move so that a given shadow is no longer path-connected.
- *Merge*: Multiple existing shadows can merge into a single shadow, when previously disconnected shadows become path-connected.

These events were originally enumerated in the context of the single-pursuer version of this problem [4] and examined more generally by Yu and LaValle [28].

A. Shadow Labels

For our pursuit-evasion problem, the crucial piece of information about each shadow is whether or not the evader might be hiding within it.

Definition *A shadow s is called* clear *at time t if, based on the pursuers' motions up to time t, it is not possible for the evader to be within s without having been captured. A shadow is called* contaminated *if it is not clear. That is, a contaminated shadow is one in which the evader may be hiding.*

Notice that, since the evader can move arbitrarily quickly, the pursuers cannot draw any more detailed conclusion about each shadow than its clear/contaminated status; if any part of a shadow might contain the evader, then the entire shadow is contaminated.

Therefore, our algorithm tracks the clear/contaminated status of each shadow. Each time a shadow event occurs, the labels can be updated based on worst case reasoning.

- *Appear*: New shadows are formed from regions that had just been visible, so they are assigned a clear label.
- *Disappear*: When a shadow disappears, its label is discarded.
- *Split*: When a shadow splits, the new shadows inherit the same label as the original.
- *Merge*: When shadows merge, the new shadow is assigned the worst label of any of the original shadows' labels. That is, a shadow formed by a merge event is labeled clear if and only if all of the original shadows were also clear.

Notice in particular that, if all of shadows are clear, then we can be certain the evader has been seen at some point. The result of this reasoning is that we can connect the shadow labels to our goal of finding a solution strategy. A pursuer strategy is a solution strategy if and only if, after its execution, all of the shadows are clear.

B. Label Dominance

The following provides some insight to the hierarchy of preferable shadow labels. Informally, we prefer one shadow

label to another if in addition to having the same shadows labelled as cleared, there are additional shadows in the label that are also labelled as cleared. This allows us to say that one shadow label *dominates* another shadow label.

Definition *Given two shadow labels corresponding to a shadow region S, we say that a label l* dominates *a label l' if the following condition holds:*

$$\forall s \in S \qquad \textbf{If} \quad l'_s = \text{clear} \quad \textbf{then} \quad l_s = \text{clear}$$

This relation is useful because our algorithm discards any shadow labels that are dominated by another shadow label reachable at the same pursuer configuration.

V. Sample-Generated Pursuit-Evasion Graph

This section introduces the primary data structure used in our algorithm. We begin by describing the graph's structure and also elaborate on a non-trivial graph operation.

A. Graph Structure

The Sample-Generated Pursuit-Evasion Graph (SG-PEG) is a *rooted* directed graph whose vertices represent joint pursuer configurations. A vertex in the SG-PEG contains

1) a joint pursuer configuration (denoted jpc), and
2) the set of non-dominated shadow labels reachable by following a path from the root, through the graph, to that configuration.

For an edge to exist between any two vertices in the SG-PEG there must be a line segment in F^n that connects the joint pursuer configuration at the source vertex with the joint pursuer configuration at the target vertex. Given an arc of the SG-PEG, $e = (x, y)$, the edge stores a mapping from the reachable shadow labels in x to the corresponding shadow labels in y.

The operations available to a SG-PEG graph are AddVertex and AddEdge. These operations differ from the corresponding operations on a standard graph because of the book-keeping needed to keep track of the reachable shadow labels. The AddVertex operation is trivial, but details concerning the AddEdge operation appear in the next section.

B. Edge Creation

When a new connection is established between a source and target vertex in the SG-PEG, the source's reachable shadow labels are used to update the target's reachable labels (Algorithm 1). In this section we discuss the shadow label update criterion, the update label subroutine, and the process of adding a new reachable label to a vertex.

1) Computing a New Label: In the authors' prior work [23], we provided a family of polynomials that capture where critical changes can occur to the region of the environment hidden from the pursuers. Although complete, the quantity and complexity of the polynomials (there are $\mathrm{O}\left(n^3m^3\right)$ polynomials, where n corresponds to the number of pursuers and m corresponds to the number of environment vertices) in this family makes the task of analytically identifying where these changes occur computationally expensive. Instead, we update the shadow labels numerically.

Algorithm 1 AddEdge(v, v')

Input: a source vertex v and a target vertex v'

1: **for each** label **in** v's reachable set **do**
2: updated $\leftarrow$ ComputeLabel(v.jpc, label, v'.jpc)
3: AddReachable(v', updated)

The general idea is that if we partition the line segment connecting any two joint pursuer configurations in F^n into a collection of evenly spaced joint pursuer configurations we can incrementally track the shadow changes. To ensure that all of the shadow events are captured there must be at least one sample capable of capturing each successive shadow event while traversing along the segment.

The computation of a new shadow label (Algorithm 2) takes as input two joint pursuer configurations, a source and target, and a shadow label corresponding to the shadow region at the source configuration. The output is the shadow label that results from the pursuers moving from the source configuration to the target configuration given the initial shadow label. Figure 3 illustrates this process. Initially, there are two contaminated shadows. As the pursuers move to the target configuration, a shadow appears as the pursuers move to the right (a cleared shadow). As the pursuers reach the target configuration the central shadow disappears.

We begin by partitioning (Algorithm 2 line 2) the segment connecting the source and target configurations in F^n into a finite collection of evenly spaced joint pursuer configurations. We then loop through this collection of joint pursuer configurations, updating the shadow label along the way, returning the final label of the sequence.

The process of computing the new shadow labels for our discretized segments appears in Algorithm 2 lines 4-11. The process starts by computing the shadow regions of both the source and target configurations. We initialize the label corresponding to the target configuration as all cleared. We check all of the shadows in the shadow region of the goal configuration for an intersection with contaminated shadows belonging to the shadow region of the source configuration. If an intersection with a contaminated shadow occurs then the corresponding shadow in the target configuration is also labelled as contaminated.

2) Adding a Reachable Label: The final step involves adding the newly computed shadow label to the target vertex (Algorithm 3). It may also be the case that the individual shadows of the new label are all cleared, in which case a solution has been found. If the target vertex contains a shadow label in its set of reachable labels that dominates the new shadow label, then the new label does not contribute any new information and we return. Similarly, if there are labels in the vertex's set of reachable labels that are dominated by the new shadow label then those labels are removed. If the new shadow label is not dominated and is not a solution strategy then we add the new shadow label to the vertex's

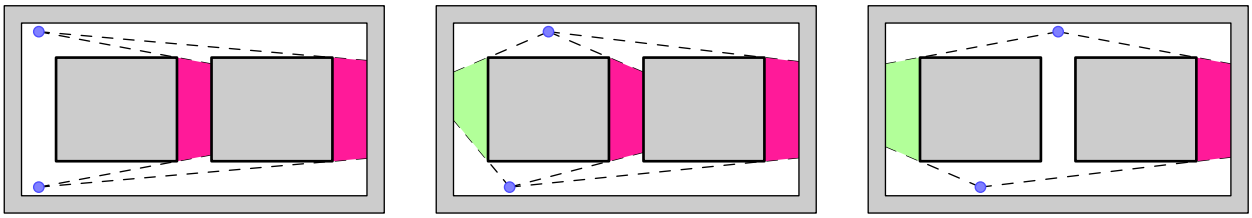


Fig. 3: An illustration of the update step. Initially there are two contaminated shadows (red). After running the COMPUTELABEL method, there is a cleared shadow (green) and a contaminated shadow (red).

Algorithm 2 COMPUTELABEL(p, l, p')

Input: a starting configuration p, starting label l, and a goal configuration p'

1: label $\leftarrow l$
2: $< p_1, \ldots, p_k > \leftarrow$ DISCRETIZE(p, p')

3: **for each** $p_i,\ p_{i+1}$ **where** $i < k$ **do**
4: oldshadows $\leftarrow$ SHADOWREGION(p)
5: newshadows $\leftarrow$ SHADOWREGION(p')
6: newlabel $\leftarrow 0 \cdots 0$ ▷ initially all cleared
7: **for each** s' **in** newshadows **do**
8: **for each** s **in** oldshadows **do**
9: **if** label$_s = 1$ **and** s' intersects s **then**
10: newlabel$_{s'} \leftarrow 1$
11: label $\leftarrow$ newlabel
12: **return** label

Algorithm 3 ADDREACHABLE(v, l)

Input: a SG-PEG vertex v and a label l

1: **function** ADDREACHABLE(v, l)
2: **if** v contains a label that dominates l **then return**

3: add l to v as a reachable label
4: delete labels in v dominated by l
5: **if** ALLCLEAR(l) **then**
6: **Output Solution** v ▷ Is l a solution?

7: **for each** out **in** Neighbors(v) **do**
8: newlabel $\leftarrow$ COMPUTELABEL(v.jpc, l, out.jpc)
9: ADDREACHABLE(out, newlabel)

reachable set. This label now permeates the graph recursively via the vertex's outgoing edges. A label is calculated for each of the vertex's neighbors, and if this label is added to the neighbors reachable set, then the process repeats itself. The process ends when no additional reachable labels are found.

Note that if a vertex does not belong to the same connected component as the root vertex then its set of reachable labels is empty. Because of the recursive nature of Algorithm 3, a vertex that serves as a bridge between the connected component containing the root vertex and another connected component will cause the reachable data to permeate through the SG-PEG.

Algorithm 4 SOLVE(p, F, A)

Input: a starting configuration p, an environment F, and an abstract sampler A

1: ADDVERTEX$(p, \{0 \cdots 0\})$
2: **while** a solution has not been found **do**
3: $s \leftarrow A$.GETSAMPLE()
4: $x \leftarrow$ ADDVERTEX(s)

5: **for each** y **in** SG-PEG vertices **do**
6: **if** $(\overline{xy} \subset F^n)$ **and**
length$(x, y) <$ maxlength **and**
cycleLength$(x, y) >$ mincycle **then**
7: ADDEDGE(x, y) ▷ Digraph edge
8: ADDEDGE(y, x) ▷ Digraph edge
9: **return** EXTRACTSOLUTION(solution)

VI. ALGORITHM

In this section we detail how our algorithm uses a SG-PEG to search for a pursuer solution strategy. Our algorithm (Algorithm 4) begins by creating a SG-PEG vertex. This vertex's joint pursuer configuration is the initial joint pursuer configuration supplied to our algorithm and it's set of reachable shadow labels contains only a single label whose shadows are all contaminated. This is the *root* vertex of our SG-PEG. We then proceed by obtaining samples in F^n, checking these samples for potential connections with existing vertices in the SG-PEG graph, and update the SG-PEG where necessary when edges are created.

A. Abstract Sampler

Our main search algorithm uses an *abstract sampler* to return a joint pursuer configuration (Algorithm 4 line 3).

Definition *An* abstract sampler *is a joint probability density function whose continuous random variables are the pursuers' positions in* F.

The only functionality that we require an abstract sampler to have is the ability to generate a point in F^n. The benefit of using an abstract sampler is that our algorithm is not dependent on a specific sampler to generate a solution strategy. This allows us to choose samplers that efficiently explore F^n. Note that the goal of catching the evaders means that the best sampling strategies may differ from those used in traditional motion planning algorithms. However, for

our algorithm to be probabilistically complete, the abstract sampler must have a support equal to F^n (Section VI-D).

We demonstrate the feasibility of using an abstract sample generator in our algorithm by providing simulation results that utilize various sample generators (Section VII).

B. Edge Criteria

In this section we discuss the constraints used in our main algorithm that determine whether an edge should connect two vertices (Algorithm 4 line 6). The three constraints can be categorized as visibility, edge length, and cycle length constraints.

1) Visibility Condition: The visibility condition states that for two vertices to share a pair of directed edges, the vertices corresponding joint pursuer configurations must be mutually visible to one another. This corresponds to the i^{th} pursuer of one configuration residing within the visibility region of the i^{th} pursuer in a neighboring configuration. Another way of interpreting this constraint is that only straight line motions are permitted between corresponding pursuers in neighboring vertices. This constraint prevents the generation of strategies in which the pursuers collide with obstacles.

2) Edge Length: To limit the amount of time spent computing the reachable data when an edge is added in the SG-PEG we place a constraint on the length of the segment connecting the vertices joint pursuer configurations in F^n. The idea is that given two joint configurations that are far apart, requiring multiple intermediary vertices as opposed to a single long connection is preferred. The intermediary vertices provide additional opportunities for any potential subsequent samples to become connected.

3) Minimum Cycle Length: To avoid an oversaturation of edges we enforce a minimum cycle length in the SG-PEG. The intuition is that if a large number of samples in F^n that are relatively close together, a large amount of resources could potentially be used computing all of the nearby transitions without necessarily revealing any new information. This optimization is aimed at minimizing the number of samples between which no shadow events occur.

C. Search for a solution strategy

The intuition is that given an initial joint pursuer configuration, we assume that all the shadows in the shadow region are contaminated. We then build a SG-PEG using an abstract sampler to select new points in F^n.

Since we maintain the reachable shadow labels during the construction of the SG-PEG, we know that a solution strategy exists if we encounter a reachable shadow label that is completely cleared. At that point we use the reachable data stored in the vertices and the shadow label mappings stored in the edges to recover a solution by following those mappings back to the root. This solution should appear as a collection of vertices in the SG-PEG. Using the joint pursuer configurations stored in the vertices as intermediary steps that the pursuers need to reach, we will have generated a joint motion strategy that is also a solution strategy.

D. Probabilistic Completeness

Finally, we argue that under certain conditions, Algorithm 4 is probabilistically complete.

Theorem 1: If the abstract sampler has a support equal to F^n, and there are no constraints on the edge length and cycle length, then our algorithm is probabilistically complete. That is, the probability of our algorithm finding a solution, if one exists, tends to 1 as the number of samples goes to infinity.

Proof Sketch: The argument proceeds in the same fashion as the probabilistic completeness proof for PRM presented by Kavraki, Kolountzakis, and Latombe [9]. The only significant difference is that, instead of considering the clearance between a solution strategy and the obstacle boundaries, we must consider the clearance from the critical boundaries at which shadow events that are not part of the final solution strategy would occur.

VII. Simulation Results

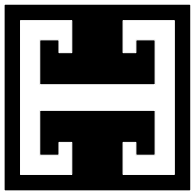
We implemented our algorithm in simulation and provide some results for three different environments, using three different sample generators, and three different cycle constraints. The environments (Figure 4) all require at least two pursuers to generate a solution strategy. As such we have deployed two pursuers to test our algorithm. The three different sample generators have the following behavior:

- SG_1 - Returns a uniform sample in F^n. This is a baseline sample generator that produces independent and identically distributed samples in F^n. This sample generator satisfies the completeness constraint.
- SG_2 - Chooses samples such that no two pursuers are mutually visible. By ensuring that the pursuers can not see one another, we attempt to maximize exploration by generating samples where the pursuers' visibility regions don't overlap. Note that this sample generator does not satisfy the completeness constraint.
- SG_3 - Selects an existing SG-PEG vertex, and for each pursuer selects a new target position from the pursuer's current visibility region. This is a local randomized sampler. By sampling within an existing SG-PEG vertex's field-of-view, we are essentially causing the search to "bloom" from the root vertex. This sample generator does not satisfy the completeness constraint.

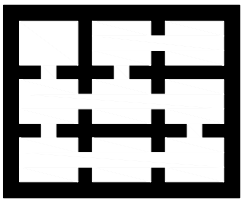
For each combination of environment, sample generator, and cycle constraints we ran 10 trials, each with a unique starting position. The simulations were implemented in C++ on a machine running Ubuntu 12.04 64-bit with an Intel Core 2 Duo E8400 processor and 4GB of RAM. Each simulation was given a maximum computation time limit of 1200 seconds. If the algorithm could not generate as solution strategy within the allotted time, we assumed that it failed.

The cycle constraints represent the extremes and one intermediary constraint. By not allowing any cycles, the SG-PEG has a tree structure, and may encounter environments where this limitation prevents our algorithm from generating a solution strategy. The other extreme has no constraint on the cycles. This means that if the samples are close together,

(a) A brick environment.



(b) An "H" environment.



(c) An office room environment.

Fig. 4: Environments used in our simulations.

then our algorithm will spend a lot of time computing reachable shadow labels as opposed to exploring.

We report a number of statistics (Tables I, II, and III) for each scenario. The first item that we report is the number of successes (Was a solution strategy found?) across all trials. For the following we report both the mean and standard deviation: the computation time in seconds, the number of SG-PEG vertices created, the number of reachable labels computed, and the total distance travelled by the pursuers.

All of the sample generators were able to produce solution strategies for the "brick" environment and had a success rate of 100% with sample generator SG_2 having the least number of vertices, reachable labels, solution length, and minimum computation time.

In the "H" environment sample generator SG_3 performed very poorly. It had only a 70% success rate when no cycles were permitted, 10% success rate for the intermediary cycle constraint, and was unable to find a solution in any of the trials when there were no cycle constraints.

In the "office room" environment sample generator SG_1 and SG_2 finally had some failures, while SG_3 continued to struggle. In this environment our algorithm was unable to generate a single solution strategy in the allotted time when no constraints were placed on the cycle length.

There are two main conclusions that we can draw from our simulations. The first is the effect the cycle length constraint has on all of the metrics that appear in Tables I, II, and III. When cycles were not allowed, the algorithm was able to generate a solution faster, requiring the pursuers to travel a shorter distance, often with a negligible increase in the number of vertices and reachable labels. When no constraints were placed on the cycle length, there was a noticeable decrease in performance. None of the samplers were able to generate a solution to the "office room" environment within the allotted time without the cycle length constraint.

The second conclusion we can draw from the simulations is the effect various samplers have on our algorithm's ability to generate a solution. The local randomized sampler (SG_3) performed poorly across all environments compared to the uniform sampler (SG_1) and the non-mutually-visible sampler (SG_2). This disparity should serve as motivation for determining what sampling strategy is most appropriate for the visibility-based pursuit-evasion problem.

No Cycles						
	SG_1		SG_2		SG_3	
success rate	100%		100%		100%	
	mean	std	mean	std	mean	std
computation time (sec)	4.63	3.77	2.28	2.31	12.84	11.19
vertices	33.90	16.78	26.90	13.01	50.80	44.75
reachable labels	23.50	16.91	12.20	9.87	56.60	46.08
solution distance (m)	78.30	32.77	55.27	22.58	58.34	20.37
Cycle Length > 15						
	SG_1		SG_2		SG_3	
success rate	100%		100%		100%	
	mean	std	mean	std	mean	std
computation time (sec)	8.76	10.13	4.46	5.55	54.14	80.72
vertices	31.10	15.60	26.90	13.01	33.80	34.46
reachable labels	22.00	16.07	12.70	10.54	43.30	39.75
solution distance (m)	99.79	74.21	61.41	37.92	78.98	62.97
No Constraints						
	SG_1		SG_2		SG_3	
success rate	100%		100%		100%	
	mean	std	mean	std	mean	std
computation time (sec)	9.51	11.83	4.83	6.26	72.17	116.21
vertices	31.10	15.60	26.90	13.01	32.90	34.83
reachable labels	21.20	16.05	12.80	10.78	43.00	40.33
solution distance (m)	88.07	50.64	60.79	36.18	80.14	76.18

TABLE I: Simulation Results for the brick environment.

VIII. Conclusion

In this paper the authors' presented a data structure called a SG-PEG that maintains the pursuers joint information state (position and reachable shadow labels). The authors' introduce an algorithm to generate a pursuer solution strategy that builds and maintains a SG-PEG. The algorithm is probabilistically complete if the sample generator used to produce samples has a support that is equal to the pursuers' joint configuration space.

There are a number of avenues for future work. Contrary to the numerical approach taken in this paper to identify shadow events, we could analytically solve the polynomials that identify changes in the shadow region.

Another avenue of potential future work centers on the sampler that we use in our algorithm. There is an abundance of research on different sampling techniques [6], [27], and it would be interesting to see the solution strategies our algorithm would generate using various sampling strategies.

Finally, we plan to investigate techniques for improving the quality of solutions, as measured by the time needed for the pursuers to execute the final solution strategy. As Figure 1 shows, paths through the SG-PEG often contain large-scale motions that appear to be unnecessary. We anticipate that a combination of both modifications to the graph construction algorithm and post-processing will be able to make substantial reductions in the path length.

Acknowledgement

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

No Cycles						
	SG_1		SG_2		SG_3	
success rate	100%		100%		70%	
	mean	std	mean	std	mean	std
computation time (sec)	57.64	30.00	141.99	290.43	411.55	354.69
vertices	96.90	47.91	380.90	777.73	276.29	198.23
reachable labels	106.30	39.37	143.20	176.06	449.71	218.49
solution distance (m)	231.57	45.19	165.58	43.94	142.24	21.96
Cycle Length > 15						
	SG_1		SG_2		SG_3	
success rate	100%		100%		10%	
	mean	std	mean	std	mean	std
computation time (sec)	145.44	76.80	209.16	318.14	685.19	0.00
vertices	95.70	48.27	380.90	777.73	75.00	0.00
reachable labels	132.10	43.82	99.00	48.61	161.00	0.00
solution distance (m)	511.61	329.79	473.10	569.87	144.42	0.00
No Constraints						
	SG_1		SG_2		SG_3	
success rate	100%		100%		0%	
	mean	std	mean	std		
computation time (sec)	177.91	120.23	182.42	114.42		
vertices	95.70	48.27	380.90	777.73		
reachable labels	120.60	45.78	91.60	39.56		
solution distance (m)	543.08	373.22	362.73	348.46		

TABLE II: Simulation Results for the H environment.

No Cycles						
	SG_1		SG_2		SG_3	
success rate	100%		100%		30%	
	mean	std	mean	std	mean	std
computation time (sec)	507.59	299.96	380.35	277.18	621.78	511.67
vertices	104.90	58.38	77.40	24.45	58.33	37.81
reachable labels	162.40	89.15	126.60	81.80	139.00	98.75
solution distance (m)	272.33	132.78	279.56	112.25	176.12	38.40
Cycle Length > 15						
	SG_1		SG_2		SG_3	
success rate	80%		90%		20%	
	mean	std	mean	std	mean	std
computation time (sec)	540.06	365.80	421.60	252.18	813.89	11.61
vertices	75.12	26.86	72.11	18.91	37.00	11.31
reachable labels	136.12	68.73	117.22	66.25	89.00	1.41
solution distance (m)	326.52	156.00	437.35	316.35	151.22	20.30
No Constraints						
	SG_1		SG_2		SG_3	
success rate	0%		0%		0%	

TABLE III: Simulation Results for the room environment.

REFERENCES

[1] W. Chin and S. Ntafos. Shortest watchman routes in simple polygons. *Discrete and Computational Geometry*, 6(1):9–31, 1991.

[2] J. W. Durham, A. Franchi, and F. Bullo. Distributed pursuit-evasion without mapping or global localization via local frontiers. *Autonomous Robots*, 32(1):81–95, 2012.

[3] B. P. Gerkey, S. Thrun, and G. Gordon. Visibility-based pursuit-evasion with limited field of view. *International Journal of Robotics Research*, 25(4):299–315, 2006.

[4] L. J. Guibas, J.-C. Latombe, S. M. LaValle, D. Lin, and R. Motwani. Visibility-based pursuit-evasion in a polygonal environment. *International Journal on Computational Geometry and Applications*, 9(5):471–494, 1999.

[5] Y. C. Ho, A. Bryson, and S. Baron. Differential games and optimal pursuit-evasion strategies. *IEEE Transactions on Automatic Control*, 10(4):385–389, October 1965.

[6] D. Hsu, T. Jiang, J. H. Reif, and Z. Sun. The bridge test for sampling narrow passages with probabilistic roadmap planners. In *Proc. IEEE International Conference on Robotics and Automation*, pages 4420–4426, 2003.

[7] R. Isaacs. *Differential Games*. Wiley, New York, 1965.

[8] V. Isler, S. Kannan, and S. Khanna. Randomized pursuit-evasion in a polygonal environment. *IEEE Transactions on Robotics*, 5(21):864–875, 2005.

[9] L. Kavraki, M. N. Kolountzakis, and J.-C. Latombe. Analysis of probabilistic roadmaps for path planning. *IEEE Transactions on Robotics and Automation*, 14(1):166–171, February 1998.

[10] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, June 1996.

[11] A. Kleiner and A. Kolling. Guaranteed search with large teams of unmanned aerial vehicles. In *Proc. IEEE International Conference on Robotics and Automation*, 2013.

[12] A. Kolling and S. Carpin. Multi-robot pursuit-evasion without maps. In *Proc. IEEE International Conference on Robotics and Automation*, pages 3045–3051, 2010.

[13] S. M. LaValle and J. Hinrichsen. Visibility-based pursuit-evasion: The case of curved environments. *IEEE Transactions on Robotics and Automation*, 17(2):196–201, April 2001.

[14] S. M. LaValle, B. Simov, and G. Slutzki. An algorithm for searching a polygonal region with a flashlight. *International Journal on Computational Geometry and Applications*, 12(1-2):87–113, 2002.

[15] N. Noori and V. Isler. Lion and man with visibility in monotone polygons. In *Proc. Workshop on the Algorithmic Foundations of Robotics*, volume 86 of *Springer Tracts in Advanced Robotics*, pages 263–278. Springer, 2013.

[16] S. Park, J. Lee, and K. Chwa. Visibility-based pursuit-evasion in a polygonal region by a searcher. In *Proc. International Colloquium on Automata, Languages and Programming*, pages 281–290. Springer-Verlag, 2001.

[17] T. D. Parsons. Pursuit-evasion in a graph. In Y. Alavi and D. R. Lick, editors, *Theory and Application of Graphs*, pages 426–441. Springer-Verlag, Berlin, 1976.

[18] S. Rajko and S. M. LaValle. A pursuit-evasion bug algorithm. In *Proc. IEEE International Conference on Robotics and Automation*, pages 1954–1960, 2001.

[19] U. Ruiz and R. Murrieta-Cid. Time-optimal motion strategies for capturing an omnidirectional evader using a differential drive robot. *IEEE Transactions on Robotics*, 21(3), June 2013.

[20] S. Sachs, S. M. LaValle, and S. Rajko. Visibility-based pursuit-evasion in an unknown planar environment. *International Journal of Robotics Research*, 23(1):3–26, 2004.

[21] J. Sgall. Solution of David Gale's lion and man problem. *Theor. Comput. Sci.*, 259(1-2):663–670, 2001.

[22] N. M. Stiffler and J. M. O'Kane. Shortest paths for visibility-based pursuit-evasion. In *Proc. IEEE International Conference on Robotics and Automation*, pages 3997–4002, 2012.

[23] N. M. Stiffler and J. M. O'Kane. A complete algorithm for visibility-based pursuit-evasion with multiple pursuers. In *Proc. IEEE International Conference on Robotics and Automation*, 2014.

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

[25] J. Thunberg and P. Ögren. A mixed integer linear programming approach to pursuit evasion problems with optional connectivity constraints. *Autonomous Robots*, 31(4):333–343, 2011.

[26] B. Tovar and S. M. LaValle. Visibility-based pursuit-evasion with bounded speed. In *Proc. Workshop on the Algorithmic Foundations of Robotics*, 2006.

[27] H. Yeh, J. Denny, A. Lindsey, S. Thomas, and N. Amato. Uniformly sampling the medial axis. Technical Report TR13-010, Texas A&M University, 2013.

[28] J. Yu and S. M. LaValle. Shadow information spaces: Combinatorial filters for tracking targets. *IEEE Transactions on Robotics*, 28(2):440–456, 2012.