Check for updates

Article

ijrr

The International Journal of Robotics Research
2017, Vol. 36(8) 923–946
© The Author(s) 2017
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/0278364917711535
journals.sagepub.com/home/ijr
SAGE

Complete and optimal visibility-based pursuit-evasion

Nicholas M Stiffler and Jason M O'Kane

Abstract

This paper computes a minimum-length pursuer trajectory that solves a visibility-based pursuit-evasion problem in which a single pursuer moving through a simply-connected polygonal environment seeks to locate an evader which may move arbitrarily fast, using an omni-directional field-of-view that extends to the environment boundary. We present a complete algorithm that computes a minimum-cost pursuer trajectory that ensures that the evader is captured, or reports in finite time that no such trajectory exists. This result improves upon the known algorithm of Guibas, Latombe, LaValle, Lin, and Motwani, which is complete but makes no guarantees about the quality of the solution. Our algorithm employs a branch-and-bound forward search that considers pursuer trajectories that could potentially lead to an optimal pursuer strategy. The search is performed on an exponential graph that can generate an infinite number of unique pursuer trajectories, so we must conduct meticulous pruning during the search to quickly discard pursuer trajectories that are demonstrably suboptimal. We describe an implementation of the algorithm, along with experiments that measure its performance in several environments with a variety of pruning operations.

Keywords

Pursuit-evasion, optimal path planning, computational geometry

1. Introduction

Pursuit-evasion algorithms, in which one group of agents, the *pursuers*, attempts to systematically locate the members of another group, the *evaders*, have a multitude of applications in robotics. For example, beyond the obvious adversarial scenarios, many kinds of search and rescue problems (Baxter et al., 2007; Calisi et al., 2007; Kleiner et al., 2013; Murphy, 2014) can be viewed as pursuit-evasion problems. Although the victims in such situations are unlikely to avoid detection actively, their movements might be erratic and unpredictable. As a result, a strategy that treats the victims as evaders is necessary to guarantee that they are found.

This paper considers a specific form of pursuit-evasion problem which requires a pursuer to locate an arbitrarily fast evader using an omni-directional field-of-view that extends to the environment boundary. The goal is to compute a search strategy for the pursuer that ensures that the evader is seen at some point during the pursuer's search.

Guibas, Latombe, LaValle, Lin, and Motwani presented a complete algorithm (which we call, for brevity, GL^{3}M) for this problem (Guibas et al., 1999), based on tracking the occluded portions of the environment in which the evader may be hiding, decomposing that environment into a finite collection of regions called conservative regions, and searching over a directed graph induced by that decomposition. However, that prior work considers only *feasibility*; it does not attempt to minimize the distance traveled by the pursuer. The contribution of this paper is to improve upon this work by describing an algorithm that is guaranteed to find a strategy that minimizes the distance traveled. Figure 1 shows an example, illustrating the difference in outputs between GL^{3}M and our new algorithm.

Our approach starts from the same decomposition as GL^{3}M, and adds three new elements.

1. First, we consider a simpler problem in which, given a path for the pursuer, we compute the shortest path that visits the same conservative regions in the same order. We solve this problem by enumerating the line segments shared between each successive pair of conservative regions visited by the original path. This ordered list of line segments forms the input to a new algorithm

[1]Department of Computer Science and Engineering, University of South Carolina, USA

Corresponding author:
Nicholas Stiffler, Department of Computer Science and Engineering, University of South Carolina, 301 Main Street, Columbia, SC 29208, USA.
Email: stifflen@cse.sc.edu

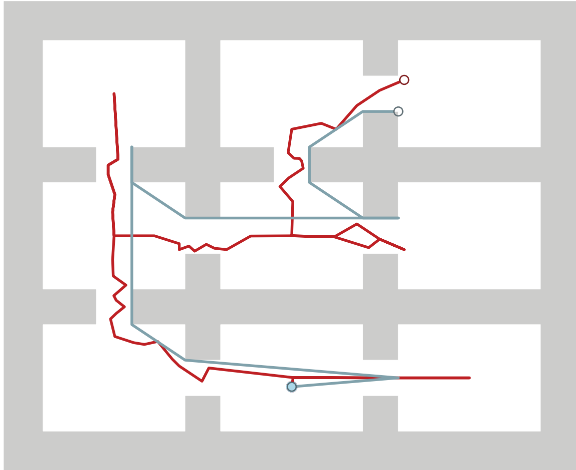


Fig. 1. The path returned by the GL^{3}M algorithm (▬) and our optimal algorithm (▬).

that computes the optimal *tour of segments*, that is, the shortest path that visits the given segments in order.
2. The tour of segments algorithm allows us to treat the pursuit-evasion problem as a search over sequences of conservative regions, rather than over the underlying continuous path space. In contrast to the vanilla breadth-first search used in GL^{3}M, our algorithm uses a forward search in which each search node represents a sequence of conservative regions. The search queue is ordered by the length of the tour of segments through the corresponding region sequence.
3. Finally, we introduce a family of *pruning operations*, that allow the search to discard region sequences that certainly cannot lead to any optimal solution. This kind of pruning is both necessary for correctness (to eliminate long sequences that repeatedly revisit the same regions without increasing the length of the tour of segments) and useful for dramatically improving the runtime of the algorithm. We provide a collection of pruning operations, ranging from very naïve to very aggressive. The resulting algorithm is complete, in the sense that if a solution exists, our algorithm will generate this path, even in cases that require recontamination of a previously searched portion of the environment.

The optimal paths generated by this algorithm are of interest because they can drastically decrease the worst-case time to capture an evader. In applications such as search and rescue, this improvement may, in principle, mark the difference between life and death for the victims. Though the computation time for GL^{3}M is less than that of our algorithm, in most reasonable circumstances, this time deficit is recovered by the decrease in execution time.

This article expands a preliminary version of this work that appeared at ICRA 2012 (Stiffler and O'Kane, 2012). This version describes the tour of segments algorithm more rigorously, including details about its internal data structures that were omitted from the conference version and provides new theoretical results, new pruning operations, and new experimental evaluations. The text has been entirely rewritten.

The structure of this paper is as follows. Section 2 reviews related work, followed by a formal problem statement in Section 3. A brief overview of GL^{3}M appears in Section 4. Section 5 presents a new result characterizing the quality of solutions the GL^{3}M algorithm produces, which motivates our work. A description of our algorithm, following the structure of the bullet list above, spans Sections 6 (tour of segments), 7 (forward search), and 8 (pruning operations). The paper closes with experimental results in Section 9 and concluding remarks in Section 10.

2. Related work

This section examines existing literature in the field of pursuit-evasion. Although this paper presents results for a visibility-based pursuit-evasion problem, we discuss the evolution of the pursuit-evasion problem from differential games (Section 2.1) to a graph-based formulation (Section 2.2) and finally to a geometric formulation (Section 2.3).

2.1. *Differential games*

The pursuit-evasion problem was originally posed in the context of differential games (Ho et al., 1965; Isaacs, 1965) and has produced an abundance of related problems. Given motion models for the pursuer and evader, the goal in these formulations is to determine the conditions necessary for them to collide in the open plane. A bound suggests that the number of pursuers required to satisfy this capture condition exceeds that needed for the visibility-based pursuit-evasion problem (Klein and Suri, 2013). More recent results have used optimal control theory to analyze differential games arising from visibility-based target tracking problems (Zou and Bhattacharya, 2016).

Several variations of pursuit-evasion games appear in differential game theory, of which two are of particular interest: the lion-and-man game and the homicidal chauffeur. In the lion-and-man game, a lion tries to capture a man who is trying to escape (Karnad and Isler, 2009; Noori and Isler, 2012, 2014; Sgall, 2001; Vander Hook and Isler, 2014). 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 (Isaacs, 1965; Ruiz and Murrieta–Cid, 2013). These problems are notable because they accentuate the two divergent ways of considering pursuit-evasion problems by considering winning conditions for each of the players (the pursuer and the evader). What conditions guarantee that a pursuer will capture the evader? What conditions guarantee that the evader can escape? Can any definitive conclusions be

drawn for particular initial conditions? This approach differs from the visibility-based formulation considered in this paper, because our "detection" criterion is satisfied when the evader lies within the pursuer's sensor footprint; we do not require the pursuer to physically capture the evader to win the game.

2.2. Graph-based formulation

A different formulation in which the domain is restricted to a discrete graph can be traced back to the independent work done by Parsons and Petrov. The motivation behind the Parsons' problem (Parsons, 1976) was the desire for a graphical model to represent the problem of finding an explorer who is lost in a complicated system of dark caves. This is one of the first formulations that considered an evader who was not necessarily adversarial. However, the worst-case assumptions provide bounds that could just as easily apply to an adversarial evader.

The Parsons' problem, also known as the edge-searching problem, is to compute a sequence of moves for the pursuers that can detect all intruders in a graph using the least number of robots. A move consists of either placing or removing a robot on a vertex, or sliding it along an edge. A vertex is considered guarded as long as it has at least one robot on it, and any intruder located therein or attempting to pass through will be detected. A sliding move detects any intruder on an edge. In relation to the current paper, it is known that for any instance of the Parsons' problem described by a planar graph, there exists an equivalent instance of the visibility-based pursuit-evasion problem considered in this paper (Guibas et al., 1999). In that sense, the problem we consider is more general.

The Parsons' problem and some of its results were later independently rediscovered by Petrov (1982) using slightly different motivating problems. Petrov's formulation considered the Cossacks and the robber game (Petrov, 1983) and the princess and the monster problem (Isaacs, 1965). Golovach showed that both problems considered an equivalent discrete game on graphs (Golovach, 1989).

There are variations of graph-based pursuit-evasion that consider both edge guarding and node guarding. One such formulation that differs from edge-searching (where searchers move across edges and guard vertices) that has a direct application to robotics is the Graph–Clear problem (Kolling and Carpin, 2010). Graph–Clear is a pursuit-evasion problem on graphs that models the detection of intruders in an environment by robot teams with limited sensing capabilities. This approach presents a plausible extension to physical robots by providing a nonintuitive way of overcoming the sensing requirements of the geometric formulation of pursuit-evasion that appears in the next section. For a more comprehensive review of recent results in graph-based pursuit-evasion we direct the reader to several relevant surveys (Abramovskaya and Petrov, 2013; Alspach, 2004; Bienstock, 1991; Borie et al., 2013; Fomin and Thilikos, 2008).

2.3. Geometric formulation

The visibility-based pursuit-evasion problem was proposed by Suzuki and Yamashita (1992) as an extension of the watchman route problem (Chin and Ntafos, 1991).[1] In this formulation, the capture condition is defined as having an evader lie within the pursuer's capture region, usually its visibility polygon or some subset thereof.

The remainder of this section reviews a portion of the substantial body of work in visibility-based pursuit-evasion, partitioned based on the number of pursuers involved in the search, organized according to results for the single pursuer and multiple pursuer variants of the problem. Though our algorithm presents an optimal strategy for the case of a single pursuer, the existing literature in multiple pursuer pursuit-evasion suggests that it is reasonable to expect our approach to be ill-fitted for this scenario.

2.3.1. Single pursuer visibility-based pursuit-evasion. We begin our discussion on single pursuer visibility-based pursuit-evasion by investigating the various models for the sensing capabilities for the pursuer. The k-searcher, first introduced by Suzuki and Yamashita, is a pursuer with k visibility beams (LaValle et al., 2002; Suzuki and Yamashita, 1992). These beams have an unlimited range that extends to the environment boundary—though they cannot see through walls—and can be freely rotated about the searcher at bounded speed independent of the pursuer's motion. This idea was later extended beyond individual rotatable beams to sensing regions where the pursuer is capable of detecting an evader that enters into its sensing region. The ∞-searcher is one such representation in which the pursuer has an omni-directional field of view (Guibas et al., 1999; Park et al., 2001). A restricted version, called the ϕ-searcher, is a pursuer whose field-of-view (Gerkey et al., 2006) is limited to an angle $\phi \in (0, 2\pi]$. Though intuition might suggest that the ∞-searcher is more powerful than the k-searcher, however Park et al. (2001) showed that any environment that is searchable by a single ∞-searcher is also searchable by a single 2-searcher (a k-searcher with $k = 2$), implying some parity of capabilities between the two. Our formulation for the pursuer's sensing capability, which appears more formally in Section 3, assumes that the pursuer is an ∞-searcher.

There are many interesting results for the single pursuer visibility-based pursuit-evasion problem that address questions of feasibility and completeness (but not optimality). The complete solution presented in GL^3M (Guibas et al., 1999) has a running time exponential in the number of inflection rays (LaValle, 2006). There exists a more complicated algorithm capable of searching an environment in time quadratic in the number of environment edges (Lee et al., 2002). This approach takes advantage of the above

result that any instance of the search solvable by an ∞-searcher can also be solved using a 2-searcher. This simplification in sensor complexity significantly reduces the apparent time complexity of the problem, though the paths generated by this algorithm are neither optimal nor nearly optimal. The algorithm presented in this paper, though more computationally intensive, guarantees to find an optimal solution, in any environment for which a solution exists. This paper is, to the authors' knowledge, the first that shows how to solve this kind of problem optimally.

An abundance of literature focuses on specific subproblems that arise in pursuit-evasion when additional constraints are placed on the environment and/or the pursuer. These subproblems often contain some nuance that necessitates approaching it differently than the previously mentioned complete algorithms. For instance, not all simply-connected polygonal environments are guaranteed to be searchable with a single pursuer. However, if we relax this constraint there are known results for generating randomized strategies (Isler et al., 2005) that are likely to capture an evader. Others have studied scenarios where there are additional constraints, such as the case of curved environments (LaValle and Hinrichsen, 2001), an unknown environment (Sachs et al., 2004), a maximum bounded speed for the pursuer (Tovar and LaValle, 2006), a finite collection of non-rotating beam sensors (Stiffler and O'Kane, 2016), or constraints on the pursuer like those of a typical bug[2] algorithm (Rajko and LaValle, 2001). Though these results address more constrained models of sensing than the present paper, they produce only feasible, rather than optimal, solutions.

2.3.2. Multiple pursuer visibility-based pursuit-evasion. Because 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. Some recent results involve using some of the pursuers as stationary sentinels while other pursuers continue with the search (Kolling and Carpin, 2009). Another approach involves maintaining complete coverage of the frontier (Durham et al., 2012). Complete (Stiffler and O'Kane, 2014a) and sampling-based (Stiffler and O'Kane, 2014b) algorithms for the multi-robot visibility-based pursuit-evasion exist that identify where critical visibility events occur amongst a team of pursuers. There are other variants of the pursuit-evasion problem where the pursuers are teams of unmanned aerial vehicles (Kleiner and Kolling, 2013). Due to the substantial increase in problem complexity over the single-pursuer form of the problem, we defer the question of optimal multi-robot visibility-based pursuit-evasion to future work.

3. Problem statement

This section formalizes the visibility-based pursuit-evasion problem considered in this paper. We begin by describing the model used to represent the environment, evader, and pursuer (Section 3.1) and then give a formal definition for the areas of the environment not visible to the pursuer, called shadows (Section 3.2). Finally, we characterize the solutions, which are shortest paths that guarantee that the pursuer will see the evader (Section 3.3).

3.1. Representing the environment, evader, and pursuer

The environment is a simply connected polygonal region, defined as a closed and bounded set $W \subset \mathbb{R}^2$, with a polygonal boundary ∂W. The boundary of the environment is composed of m vertices, each of which may be either convex (interior angle less than 180 degrees) or reflex (interior angle greater than 180 degrees).

The evader is modeled as a point that can translate within the environment. Let $e(t) \in W$ denote the position of the evader at time $t \geq 0$. The path e is a continuous function $e : [0, \infty) \rightarrow W$, in which the evader can move arbitrarily fast (i.e. a finite, unbounded speed) within W. The evader trajectory e is unknown to the pursuer. Without loss of generality, we assume that there is exactly one evader. If the pursuer can guarantee the capture of a single evader, then the same strategy can locate multiple evaders, or confirm that no evaders are present.

A pursuer moves to locate the evader. We assume that the pursuer knows W. Therefore, from a given start position, the pursuer's motions can be described by a continuous function $p : [0, \infty) \rightarrow W$, so that $p(t) \in W$ denotes the position the pursuer at time $t \geq 0$. The function p is called a *motion strategy* for the pursuer. Without loss of generality, we assume that the pursuer moves with maximum speed 1.

The 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 $q \in W$, let $V(q)$ denote the visibility region at point q, which consists of the set of all points in W that are visible from point q. That is, $V(q)$ contains every point that can be connected to q by a line segment in W. Note that $V(q)$ is a closed set.

For any $q \in W$, consider the boundary of $V(q)$. The edges of this boundary are either along ∂W or belong to an occlusion ray.

Definition 1. An *occlusion ray*, $\overrightarrow{qr}$, is a ray starting at the pursuer position q and tangent to a visible environment reflex vertex r.

Informally, an occlusion ray originating at point q is a ray that acts as a boundary separating a visible and non-visible portion of W.

The time of capture for an evader following trajectory e and a pursuer executing motion strategy p is defined as

$$t_c(p, e) = \min\{t \geq 0 \mid e(t) \in V(p(t))\}$$

The pursuer's goal is to capture the evader regardless of the evader's trajectory.

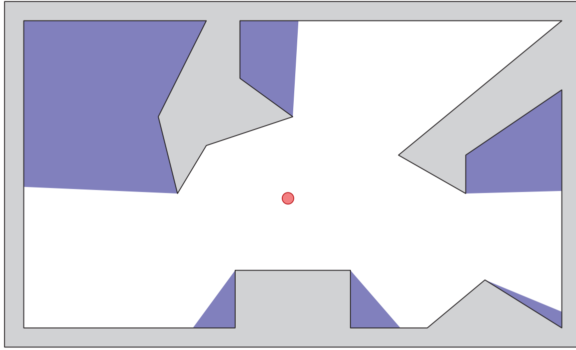


Fig. 2. An environment with a pursuer (red circle) and six shadows (filled path-connected regions).

Definition 2. A pursuer motion strategy p is a *solution strategy* if there exists a finite time of capture, denoted $t_c(p)$ and defined as

$$t_c(p) = \max_e \ t_c(p, e)$$

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

$$p^* = \operatorname*{argmin}_p \left(t_c(p)\right)$$

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

3.2. Shadows.

The key difficulty in locating the evader is that the pursuer cannot, 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 pursuer at any time.

Definition 3. The portion of the environment not visible to the pursuer at time t is called the *shadow region* $\mathcal{S}(t)$, and defined as

$$\mathcal{S}(t) = W - V(p(t))$$

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

Definition 4. A *shadow* is a maximal path-connected component of the shadow region.

Notice that $\mathcal{S}(t)$ 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.

3.2.1. Shadow labels. For our pursuit-evasion problem, the crucial piece of information about each shadow is whether the evader might be hiding within it.

Definition 5. A shadow s is called *cleared* at time t if, based on the pursuer's motions up to time t, it is not possible for the evader to be within s without having been captured.

Definition 6. A shadow is called *contaminated* if it is not clear. That is, a contaminated shadow is one in which the evader may be hiding.

We can assign a binary label to each shadow corresponding to the cleared/contaminated status of the shadow. A label of 0 means that the shadow is cleared and similarly, a label of 1 means that the shadow is contaminated. Notice that, since the evader can move arbitrarily quickly, the pursuer 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.[3] Using this worst-case reasoning, we can completely represent the pursuer's progress in searching for the evader by its current configuration and the current shadow labels.

3.2.2. Shadow events. As the pursuer moves, the shadows can change in any of five ways, called *shadow events*.

1. *Appear*: A new shadow can appear, when a previously visible part of the environment becomes hidden.
2. *Disappear*: An existing shadow can disappear, when the pursuer moves to a location from which that region is fully visible.
3. *Split*: A shadow can split into multiple shadows, when the pursuer moves in such a way that a given shadow is no longer path-connected.
4. *Merge*: Multiple existing shadows can merge into a single shadow, when previously disconnected shadows become path-connected.
5. *Push*: An existing shadow can be pushed between pairs of neighboring environment reflex vertices, when the pursuer's motion changes the cardinality of the set of visible environment reflex vertices.

These events, which are illustrated in Figures 3 to 5, were originally enumerated in the context of the single-pursuer version of this problem (Guibas et al., 1999) and examined more generally by Yu and LaValle (2012).

Assuming that the vertices of the environment are in general position[4], we need only be concerned with a single shadow event occurring at a given moment. However, if the environment vertices are not in general position, multiple events may occur simultaneously. The above update rules still apply, but this may require extra bookkeeping to ensure that the shadow labels are correctly accounted for. As an example, consider Figure 6, where a split occurs and one of the shadows is instantaneously pushed.

3.2.3. Label update rules. Each time a shadow event occurs, the labels can be updated based on worst case reasoning. Below we describe the update rules for a shadow's label according to the shadow event that has occurred. Each

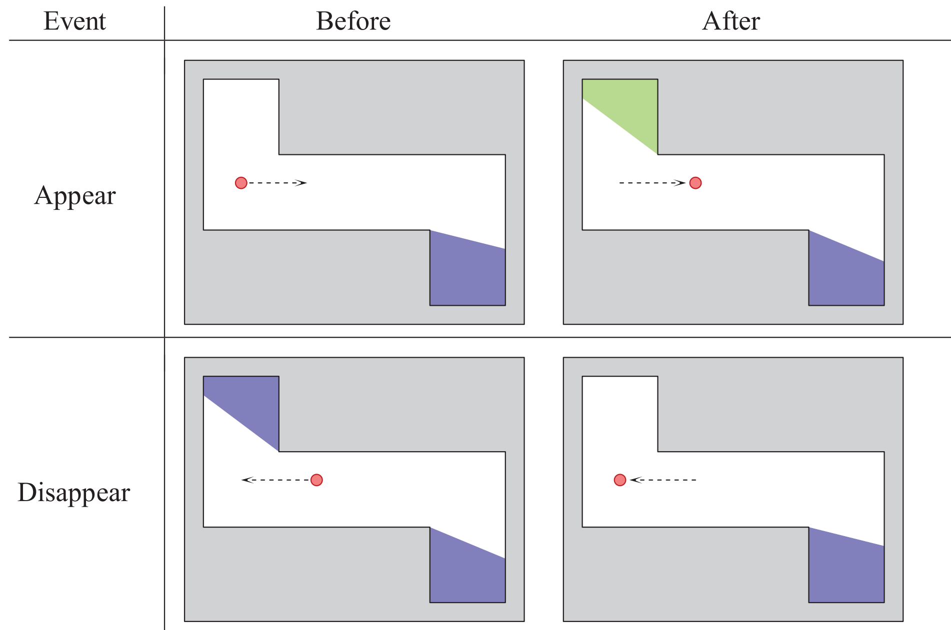


Fig. 3. An appear event increases the number of shadows by one, and the new shadow is labelled clear (green region). A disappear event decreases the number of shadows; its label is discarded.

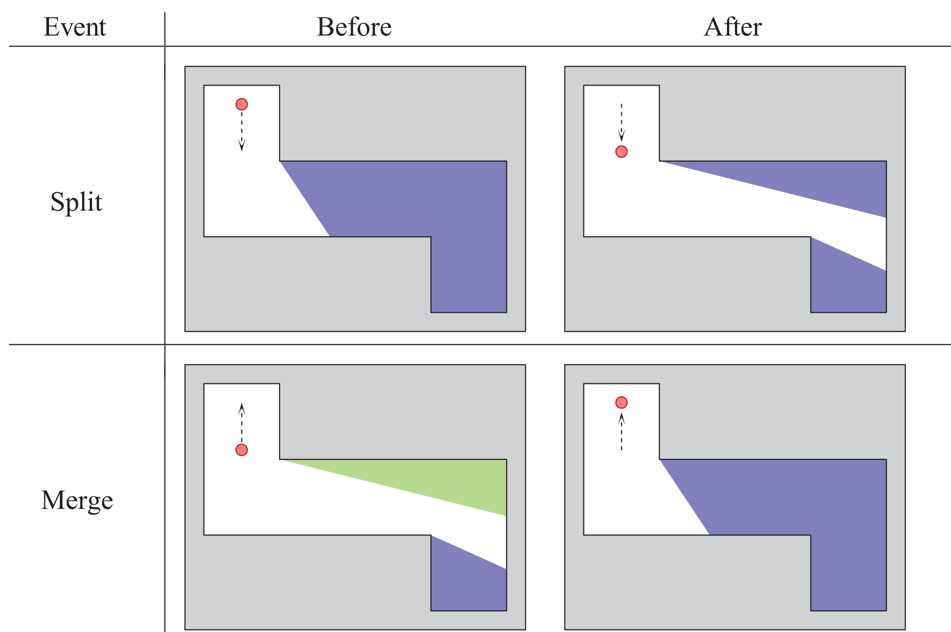


Fig. 4. When a shadow splits into multiple shadows, they inherit the same label as the original shadow. When a merge event occurs, the new shadow is clear if and only if all the original shadows are also clear.

rule describes how a label preceding the shadow event is updated immediately following a given shadow event.

1. *Appear*: New shadows are formed from regions that had just been visible, so they are assigned a clear label.
2. *Disappear*: When a shadow disappears, its label is discarded.
3. *Split*: When a shadow splits, the new shadows inherit the same label as the original.
4. *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 the original shadows were also clear.
5. *Push*: When a shadow is pushed, it maintains its current label.

Figures 3 to 5 illustrate the shadow label update rules where cleared shadows are represented as the filled

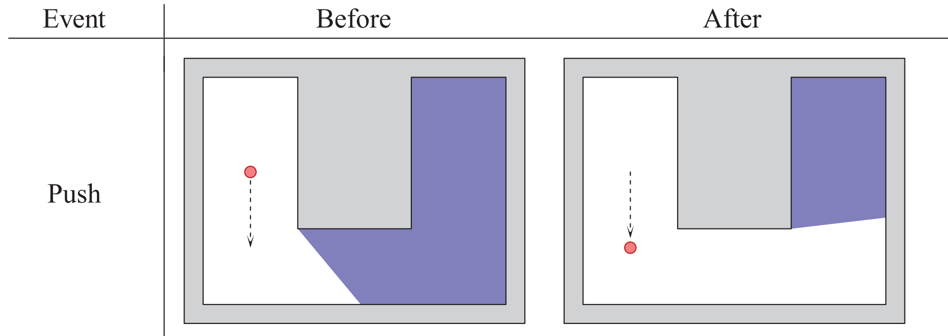


Fig. 5. A push event occurs when a shadow gets pushed between neighboring pairs of environment reflex vertices.

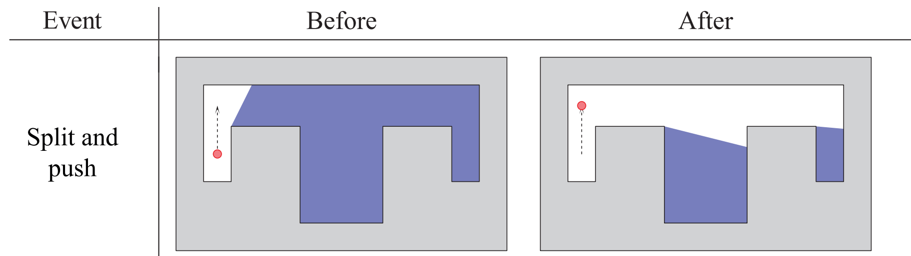


Fig. 6. A scenario where multiple events occur simultaneously. To ensure that the correct labels are computed it may be necessary to evaluate all permutations of the events. This occurs because the update rules are applied successively one after another, and an improper ordering may result in an incorrect final labeling. In the above scenario, a correct final labeling can result by first processing the split event followed by the push event.

path-connected green regions and contaminated shadows are represented as the filled path-connected purple regions.

3.3. Objective

We can incorporate this idea of reasoning about evaders via shadows to reformulate the pursuer's goal in terms of shadows rather than evader positions. Recall the definition of solution strategy from Definition 2 where the pursuer's goal was stated as computing a finite time of capture for an evader over all possible evader trajectories. Using the definitions of cleared and contaminated from above to describe a shadow's current status, we know that if all the shadows in the shadow region are cleared, 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.

Definition 7. A pursuer motion strategy is a *solution strategy* if and only if it reaches a pursuer configuration in finite time in which all the shadows are cleared.

We now have two distinct but equivalent definitions of a solution strategy.

4. GL^{3}M

The prior work of Guibas, Latombe, LaValle, Lin, and Motwani is integral to understanding the techniques in this paper. As such, this section summarizes their algorithm.

The main idea behind that work is a blueprint for changing the continuous problem of finding a pursuer trajectory into a simpler discrete problem. Recall that the goal is to compute a solution strategy for the pursuer (Definition 2). The GL^{3}M algorithm considers the boundaries in the environment that induce a change to the shadow region. This view allows them to use the alternative definition of solution strategy from Definition 7. The problem can now be rephrased as finding a sequence of shadow events that can guarantee that the evader is captured. Once a valid sequence has been found, constructing the pursuer's trajectory is trivial.

In the remainder of this section we investigate how pursuer motions generate shadow events (Section 4.1), and describe a graph structure and algorithm for solving the single pursuer visibility-based pursuit-evasion problem (Section 4.2). The succeeding section (Section 5) presents a new result, proving that for any arbitrary factor, an environment exists for which the solutions generated by this algorithm are longer than the optimal solutions by this factor.

4.1. Critical information changes

During the execution of a strategy, the pursuer must identify the contaminated shadows in the shadow region. This piece of information is dependent upon the initial position of the pursuer and the pursuer's history of past positions, up to the current time. As the pursuer moves, this information changes continuously; however, to develop a complete

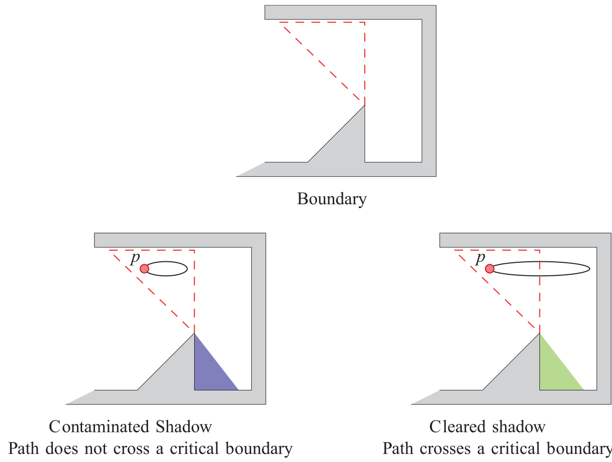


Fig. 7. An illustration of the concept of conservative regions.

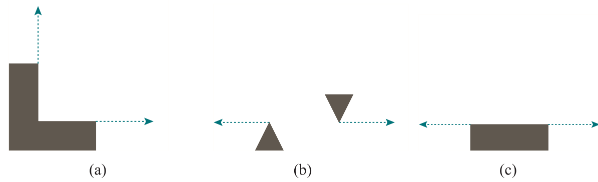


Fig. 8. Ray shooting is performed for three general cases to form the conservative regions.

algorithm, the authors need only be interested in tracking those instances where the pursuer's information changes combinatorially. That is, we are only concerned with pursuer movements that generate shadow events, as seen in Figure 7.

Definition 8. A region $R \subseteq W$ is a *conservative region* if every continuous path that remains within R fails to generate any shadow events.

By definition, a conservative region has the following information-conservation property: while the pursuer remains within a conservative region, the pursuer's shadow labels will not change.

The original paper GL^{3}M describes a visibility cell decomposition of the environment that captures where changes to the shadow region occur. As mentioned previously, a shadow event represents a change in the pursuer's knowledge about where the evader may be hiding. 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 shadow labels change.

There are five shadow events that can occur at a critical event boundary that cause a change in the pursuer's shadow labels as it traverses between conservative regions. These events (appear, disappear, split, merge, and push) were mentioned earlier in Section 3.2.

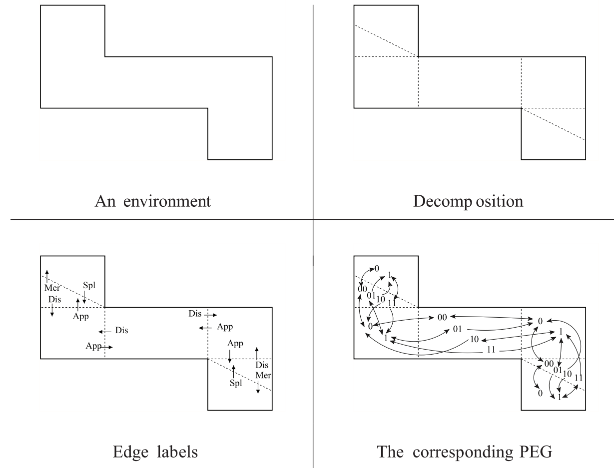


Fig. 9. An example of the Pursuit Evasion Graph for a given environment.

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

Figure 8 illustrates the various partitioning operations.

The key idea is that the regions of W induced by these ray extensions are conservative regions; that is, these rays represent the only places at which shadow events can occur.

Two conservative regions in this decomposition are called *adjacent* if they share an edge. An ordered sequence $r_1, \ldots, r_n$ of conservative regions is called *successively adjacent* if each successive pair (r_i, r_{i+1}) is adjacent.

4.2. The pursuit-evasion graph

With this information, the algorithm constructs a directed graph, as shown in Figure 9.

Definition 9. The *pursuit-evasion graph* (PEG) is a directed graph composed of nodes that contain a shadow label and a reference to a conservative region. A node exists in the PEG for each possible shadow label combination for every conservative region. Its edges are the set of shadow events that occur from crossing an event boundary from one adjacent conservative region to another.

For a given PEG-node v, we write $\mathrm{cr}(v)$ and $\mathrm{label}(v)$ to denote the conservative region and shadow label associated with v, respectively. We use the term *PEG-path* to refer to a path through the PEG.

The GL^{3}M algorithm starts at the PEG-node that contains $p(0)$ with a shadow label of $1\cdots1$. 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 shadow label of $0\cdots0$. This PEG-path provides a sequence of conservative regions to visit. The algorithm then constructs a path through W by moving to the centroid of each conservative region that appears in the sequence.

5. Analysis of path length for GL^{3}M

The algorithm described in GL^{3}M employs a breadth-first search to search the PEG. As a direct result of the breadth-first search, the returned path minimizes the number of conservative regions visited by the pursuer. The authors make no claims about the length of the pursuer's path. The following new result establishes that GL^{3}M can produce paths of arbitrarily poor quality.

Theorem 1. *For any $\epsilon > 0$, there exists an environment W_ϵ for which the path generated by GL^{3}M is at least $1+\epsilon$ times longer than the optimal path.*

Proof. Given $\epsilon > 0$, let $a = 1+\epsilon$. Define W_ϵ as an 8-vertex environment, with vertices $(0,-1)$, $(2a,1)$, $(0,3)$, $(a,4)$, $(0,5)$, $(-2a,3)$, $(0,1)$, and $(-a,0)$. Figure 10 illustrates this construction for two different values of ϵ.

Starting from $p(0)=(0,0)$, the shortest solution strategy moves directly upward 3 units, terminating at $(0,3)$. The output from GL^{3}M, however, visits the centroids of four conservative regions. These centroids lie at $(a/3,0)$, $(a,1)$, $(0,2)$, and $(-a,3)$. Including the travel from the initial position to the first centroid, this path has length

$$d = \frac{a}{3} + \sqrt{\frac{4a^2}{9} + 1} + 2\sqrt{a^2+1}$$

We can bound this from above by

$$d \geq \frac{a}{3} + \sqrt{\frac{4a^2}{9}} + 2\sqrt{a^2} = \frac{a}{3} + \frac{2a}{3} + 2a = 3a = 3(1+\epsilon)$$

Recalling that the optimal path has length 3, we conclude that GL^{3}M generates a path in W_ϵ that is at least $1+\epsilon$ times longer than optimal. □

The upshot of this theorem is that the amount of sub-optimality produced by GL^{3}M is not negligible, and in fact cannot be bounded. This difference motivates our new algorithm, introduced in the following sections.

6. Optimal tours of segments

We now turn our attention to our new algorithm for generating pursuer strategies that minimize the distance travelled. We begin, in this section, by describing the subroutine we use to solve the subproblem of computing the shortest path that traverses a given sequence of conservative regions.

The next theorem motivates our interest in this problem, by making a connection between the concept of a solution strategy for the pursuer and the sequence of conservative region boundary edges crossed by the pursuer while executing that strategy.

Theorem 2. *For any solution strategy γ, let $(r_1,\ldots,r_n)$ denote the successively adjacent sequence of conservative regions visited by γ. Let $(s_1,\ldots,s_{n-1})$ denote the sequence of conservative region boundary edges crossed by γ, in which s_i is the shared boundary segment between the regions c_i and c_{i+1}. Then any other pursuer trajectory γ' that visits $(s_1,\ldots,s_{n-1})$ in the same order, without crossing any other conservative region boundaries, is also a solution strategy.*

Proof. This is a direct consequence of the definition of conservative region. □

See Figure 11. Based on this result, which ties the notion of a solution strategy closely to the notion of a successively adjacent sequence of conservative regions, we consider the following problem (deferring, to Sections 7 and 8, the question of how to select a such sequence of conservative regions).

Definition 10. Given a point p and an ordered collection of segments $(s_1,\ldots,s_{n-1})$, the shortest path that starts at p and visits the segments $(s_1,\ldots,s_{n-1})$ in order is called a *tour of segments*, denoted $\mathrm{ToS}(s_1,\ldots,s_{n-1})$.

When there is no ambiguity, we occasionally abuse this notation by giving a successively adjacent sequence of conservative regions or a PEG-path (instead of a sequence of segments) as the parameter to ToS, to refer to the tour of segments for the sequence of shared boundary segments.

This section presents an algorithm for computing such tours of segments. Dror et al. (2003) showed how to compute tours in a similar scenario, in which the intermediate steps are polygons rather than segments. 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 (Section 6.1). Second, we use a series of point location queries (Section 6.2) on these SPMs to extract the optimal tour (Section 6.3).

6.1. Shortest path maps

A *shortest path map* (SPM) is a data structure used to perform shortest path queries with the requirement that the path visit a segment s along the way. Given a start point p and a segment s, a SPM can be constructed which partitions the plane into four two-dimensional cells, five one-dimensional cells, and two zero-dimensional cells. Figure 12 shows an example. The key idea is that all shortest paths

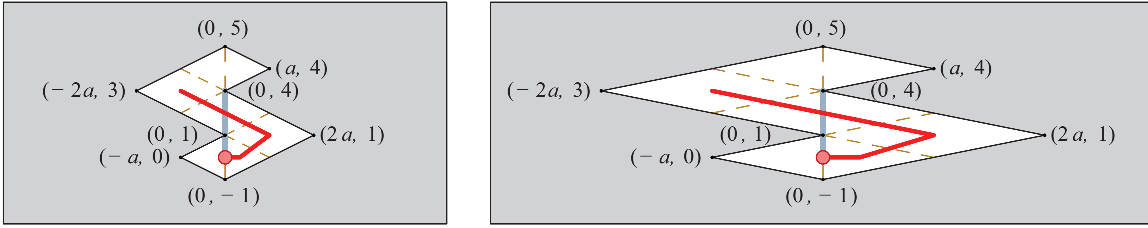


Fig. 10. Examples of the construction in the proof of Theorem 1. Conservative region boundaries are shown as dashed lines. The optimal solution strategy is shown in . The path generated by GL^{3}M is in . [left] $\epsilon = 1$, $a = 2$. [right] $\epsilon = 4$, $a = 5$.

starting from p to all points in one of the aforementioned cells will have an equivalent combinatorial structure.

SPM: zero-dimensional cells The two 0D cells in a SPM correspond to the two endpoints of segment s, left(s) and right(s).

SPM: one-dimensional cells There are five 1D cells in a SPM, denoted A, B, C, D, and E. The 1D cells are constructed from segment s and the start point p. One of these 1D cells is an open line segment and corresponds to the interior of s. The four remaining 1-cells are all open rays, two originating from left(s) and two originating from right(s). The following table describes each of the 1D cells.

Line segment A	Segment s.
Ray B (upper left ray)	A ray originating from left(s) (the left endpoint of s) in the direction left(s) $-p$.
Ray C (lower left ray)	A reflection of ray B over the line segment s.
Ray D(upper right ray)	A ray originating from right(s) (the right endpoint of s) in the direction right(s) $-p$.
Ray E(lower right ray)	A reflection of ray D over the line segment s.

SPM: two-dimensional cells There are four 2D cells in a SPM that are separated by the 1D cells. The following describes which 1-cells form the boundary of our 2D cells.

6.2. *Queries in a shortest path map*

Using this structure, and given a query point q, we can compute the shortest path from p to q via s, as shown in Figure 13. There are four general cases which correspond to the 2D cells in of our SPM.

(a) If q is in region $R1$, then the shortest path from p to q via s is a "left turn" at the left endpoint of s.

Region $R1$	The region of the plane between ray B, left(s), and ray C.
Region $R2$	The region of the plane between ray B, left(s), segment A, right(s), and ray D.
Region $R3$	The region of the plane between ray D, right(s), and ray E.
Region $R4$	The region of the plane between ray C, left(s), segment A, right(s), and ray E.

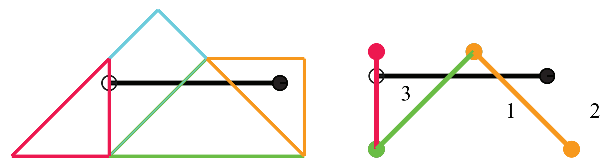


Fig. 11. [left] A sequence of four conservative regions to be visited in order from right to left, along with the corresponding shortest path. [right] The shortest path can be found by considering only the shared boundary segments.

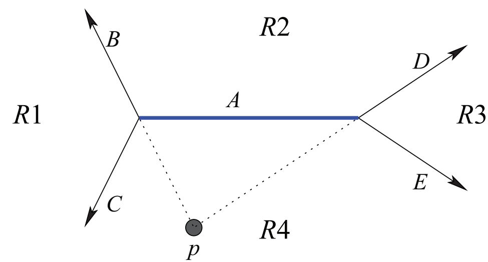


Fig. 12. A single shortest path map. These four rays and one segment subdivide the plane into regions with combinatorially equivalent shortest paths.

(b) If q is in region $R2$, then the shortest path from p to q via s is to go "through" s directly to q.
(c) If q is in region $R3$, then the shortest path from p to q via s is a "right turn" at the right endpoint of s.
(d) If q is in region $R4$, then the shortest path from p to q via s is to "bounce" off s.

We have described the procedure for creating a single SPM, however when computing multiple SPMs for a

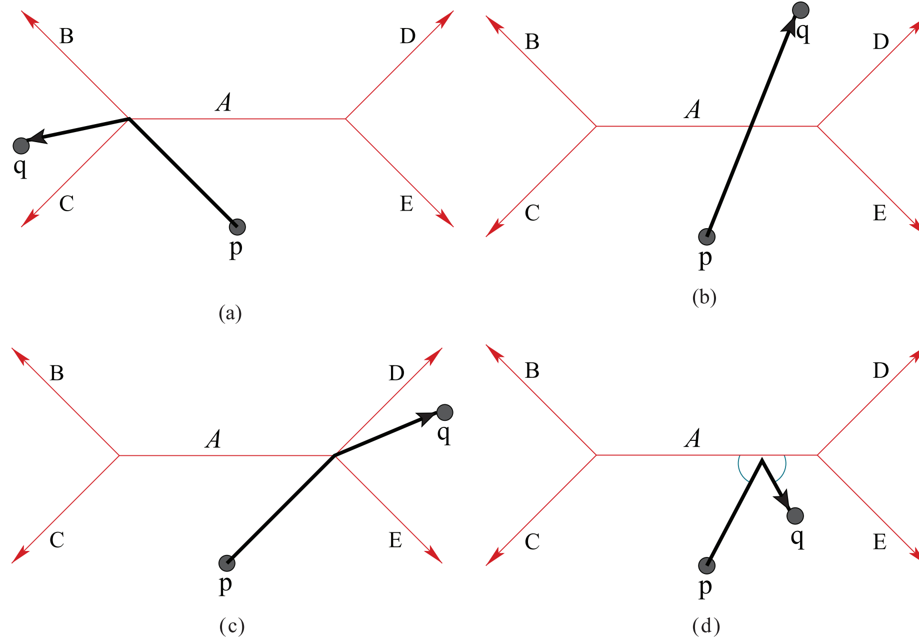


Fig. 13. The SPM for the first segment s divides the plane according to the combinatorial structure of the shortest path from p to s to a query point q.

sequence of segments we will need a more general construction that has two start points p_L and p_R which are determined by point location queries in the previous SPMs, as shown in Figure 14(a). The construction is like the construction of a single SPM 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 k-dimensional cell 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 14(b) illustrates this process.

Algorithm 1 SELECTSTARTPOINT(i, q)

Input: An index i for a specific SPM and a query point q

```
1:  if i = 0 then
2:  |   return p
3:  end if

4:  r ←LOCATE(i − 1, q)
5:  switch ( r )
6:  |   case R1 or B or C :
7:  |   |   return left(s_{i−1})
8:  |   case R2 or A or left(s) or right(s) :
9:  |   |   return SELECTSTARTPOINT(i − 1, q)
10: |   case R3 or D or E :
11: |   |   return right(s_{i−1})
12: |   case R4 :
13: |   |   return SELECTSTARTPOINT
        (i − 1, REFLECT(q, s_{i−1}))
14: end switch
```

6.3. 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-1} and all of the SPMs. This produces a subdivision of s_{n-1} 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-1}. 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

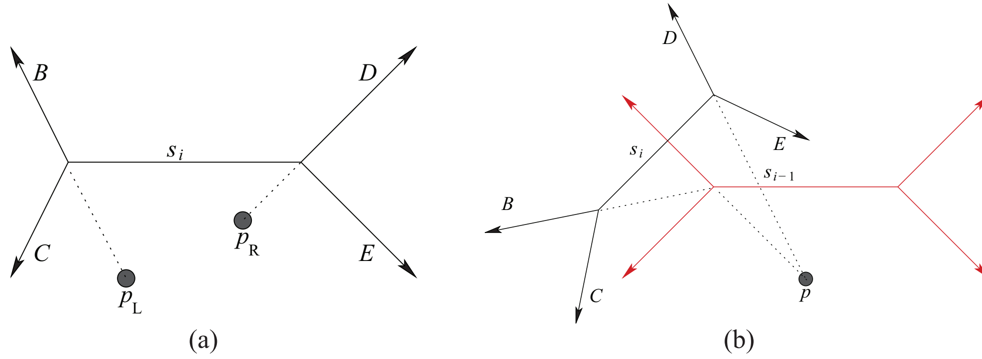


Fig. 14. Computing the shortest path map for segment s_i depends on the shortest path map for segment s_{i-1}.

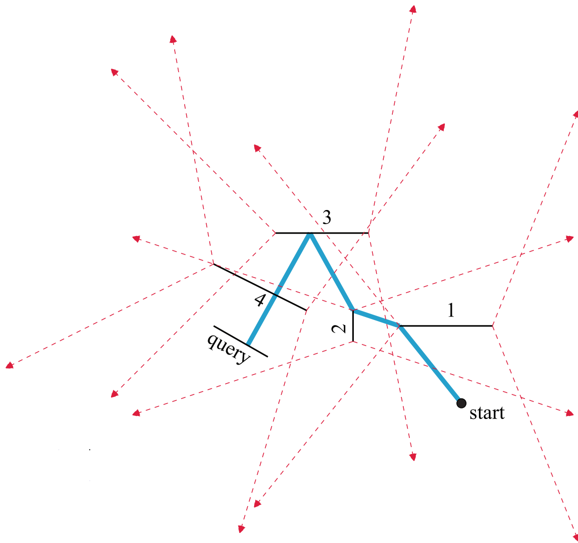


Fig. 15. A non-trivial tour constructed by visiting segments 1, 2, 3, and 4 in order before reaching the query segment. From the start point, the optimal tour takes a left turn at segment 1, a right turn at segment 2, bounces off segment 3, and then travels through segment 4 to reach the query segment.

Algorithm 2 EXTRACTPATH(i, q)

Input: A SPM index i, and a query point q

Data: A list *tour* which stores points along the optimal tour

```
1:  if i = 0 then
2:  |  tour.insert(start_pt)
3:  |  return tour
4:  end if

5:  r ← LOCATE(i, q)
6:  switch ( r )
7:  |  case R1 : B : C :
8:  |  |  tour ← EXTRACTPATH(i − 1, left(s_i))
9:  |  |  return tour.insert( left(s_{i−1}))
10: |  case R2 : A : left( s ) : right(s) :
11: |  |  tour ← EXTRACTPATH(i − 1, q)
12: |  case R3 : D : E :
13: |  |  tour ← EXTRACTPATH(i − 1, right(s_i))
14: |  |  return tour.insert( right(s_{i−1}))
15: |  case R4 :
16: |  |  reflect_pt ← REFLECT( q, s_i)
                                ▷ reflect point across segment
17: |  |  tour ← EXTRACTPATH(i − 1, r)
18: |  |
19: |  |  ▷ calculate "bounce" point
20: |  |  bounce_pt ← LINEINTERSECTION
    (s_i, ( r, tour.back( ) )
21: |  |  return tour.insert( bounce_pt)
22: end switch
23:
24: return tour
```

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.

Computing the optimal tour to reach the nth segment takes $\mathcal{O}(n^2)$. Segment s_n can potentially be partitioned a linear number of times via the SPMs for segments $s_1, \ldots, s_{n-1}$. Then for each subsegment of s_n, the algorithm may need to scan back through all $\mathcal{O}(n)$ SPMs. Usually, this computation can be performed faster, as we expect to hit a left or right endpoint prior to recursing all the way back to the start. A non-trivial example for computing the tour of segments appears in Figure 15.

The correctness of the algorithm can be established by induction on the number of segments n. For the base case of a single segment, the generated path is simply the shortest straight line segment connecting the start point to the query segment. For the inductive step, assume for the induction that the algorithm correctly solves problems of size $n-1$, and note that both SELECTSTARTPOINT and EXTRACTPATH recursively solve a family instances of size $n-1$, considering each potential start point induced by the SPMs.

7. Algorithm description

This section introduces the main body of our algorithm for the problem introduced in Section 3. Algorithm 3 summarizes the approach.

Starting from the same cell decomposition and PEG as GL^{3}M (recall Section 4), we perform a branch and bound forward search (LaValle, 2006) in which each search node corresponds to a PEG-path. (The more obvious, simpler approach in which each search node is a single PEG-node, rather than a PEG-path, would be incorrect because the additional cost of adding a new PEG-node to the end of a partial plan is not necessarily additive.)

The search begins with a one-element path, containing only the PEG-node which corresponds to the pursuer's initial position, with these shadows contaminated. Then for each search node, we determine the boundary segments crossed by visiting the underlying PEG-nodes in the given order, and then use Algorithm 2 to compute the ToS for that segment sequence. The forward search uses a priority queue of search nodes ordered the length of ToS, with the search nodes with the shortest tours extracted first.

At each iteration, the PEG-path $\hat{v} = (v_1, \ldots, v_n)$ at the head of the priority queue is extracted. This corresponds to a path from the root PEG-node to the current PEG-node v_n. New PEG-paths are generated by iterating over all of v_n's outgoing directed edges (Algorithm 3 Line 10) and appending the target of the directed edge to the current PEG-path (Algorithm 3 Line 11). Recall from Section 4.2 that these directed edges correctly account for label updates that appear in Section 3.2.3, including any shadows that might be cleared or recontaminated. Before adding a new PEG-path to the priority queue, we apply a pruning operation (PRUNABLE) that determines whether the PEG-path can be safely discarded. This pruning is essential to the operation of the algorithm; details about how it can be performed appear in Section 8.

The termination conditions for our algorithm are twofold. First, if the priority queue becomes empty, the search terminates and reports that no solution exists. Second, if the head of our priority queue ever corresponds to a PEG-path ending at a PEG-node whose shadow labels are all clear ($0 \cdots 0$), then we know that no additional expansions will generate a shorter solution strategy, so the search terminates successfully, returning the ToS of this sequence as the optimal solution strategy.

Figure 16 shows a simple example environment. Correct solution strategies in this environment clear both the left and right sides, and optimality requires the pursuer to choose which of these two tasks to complete first. The figure, which shows solution strategies computed by Algorithm 3 for a variety of starting positions, illustrates that the algorithm can make this distinction correctly: Starting points on the left search the left side first, and vice versa.

The correctness of this approach derives from Theorem 2. The termination condition for the search guarantees that any PEG-path that is not considered has either (a) been pruned by the PRUNABLE test, indicating that it cannot be part of an optimal solution, or (b) has a longer ToS than the generated solution. We are assured, therefore, that there does not exist another PEG-path corresponding to a shorter solution strategy.

Algorithm 3 FORWARDSEARCH(p)

Input: a start point p, a pruning unary operator *prune*
Data: a priority queue *pq* for PEG-paths ordered by length of the ToS

```
1:  pq.insert(GETROOT(p))
                              ▷ start with single contaminated node
2:  while not pq.empty() do
3:      v̂ ← pq.top() ▷ top PEG-path in the pq
4:      v_n ← last(v̂) ▷ PEG-node reached by following v̂
5:      if label(v_n) = 0···0 then        ▷ test for a solution
6:          return v̂
7:      end if
8:
9:      ▷ Outgoing directed edges from PEG-node
10:     for each out ∈ OUTGOINGNODES(v_n) do
11:         path ← (v_1, ..., v_n, out)
                                            ▷ append node
12:         if not PRUNABLE(path) then
13:             pq.insert(path)
                                  ▷ add new PEG-path to pq
14:         end if
15:     end for
16: end while
17:
18: return NO SOLUTION
```

8. Sequence dominance and pruning strategies

This section expounds upon the pruning strategy employed by Algorithm 3. The intuition is that we want the algorithm to discard any partial solutions that we can guarantee do not serve as a prefix to the optimal solution. We describe six options of varying complexity and effectiveness for realizing this test. These six pruning strategies can be understood in three groups.

1. The first strategy, UNAVAILING pruning, trivially declines to prune any PEG-paths. It serves to motivate the need for pruning by showing that the search generally cannot succeed, in any length of time, if pruning is not considered during the search.
2. The next two strategies, CYCLE-FREE and REGRESSION, are related in that they consider specific properties of a single PEG-path when determining whether that PEG-path should be added to the priority queue. The idea is to identify redundancies (in the case of

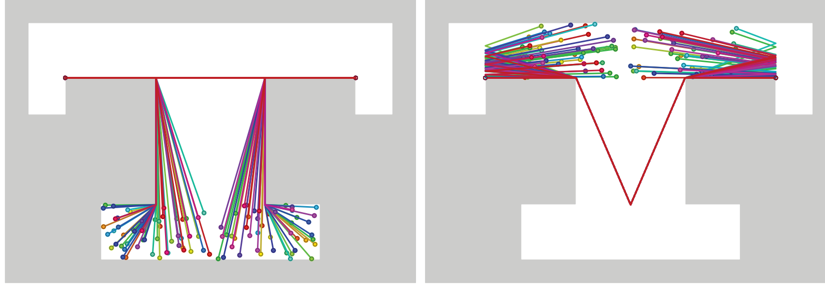


Fig. 16. A collection of 200 optimal pursuer strategies, computed by our approach, for various starting points within a simple environment. Notice that the algorithm correctly determines whether the shortest path should start by moving to the left or to the right.

Name	Description	Section
Unavailing	No pruning.	8.1
Cycle-Free	Prunes PEG-paths that contain multiple instances of the same PEG-node.	8.2
Regression	Prunes PEG-paths that visit a conservative region a second time with different but “worse” shadow labels.	8.3
Path Dominance (Quadrilateral)	Prunes PEG-paths that cannot lead to a shorter solution than some other known PEG-path, by a direct comparison of endpoints.	8.4
Path Dominance (Endpoint)	Prunes PEG-paths that cannot lead to a shorter solution than some other known PEG-path, by appending the final segment of one path to the end of the other.	8.5
Path Dominance (Lookahead)	Prunes PEG-paths that cannot lead to a shorter solution than some other known PEG-path, by examining each of the possible exit segments from the final conservative region.	8.6

Fig. 17. Preview of the pruning strategies discussed in this section to reduce the number of PEG-paths that our algorithm must consider. These are ordered in increasing order of both computation time and pruning effectiveness.

CYCLE-FREE) or regressions (in the case of REGRESSION) in the PEG-nodes visited by the path.

3. The remaining three strategies, PATH DOMINANCE, ENDPOINT, and LOOKAHEAD pruning, are based on comparisons between pairs of PEG-paths. They seek to identify conditions under which a given PEG-path is provably inferior to some other PEG-path terminating in the same conservative region. Each one maintains a list of non-dominated PEG-paths for each conservative region. These lists are updated as the search proceeds. These three pruning strategies differ in the specifics of how path dominance is confirmed, generally trading extra computation time for more aggression in the pruning.

A preview appears in Figure 17.

8.1. Unavailing pruning

A naïve approach to implementing the forward search is to ignore the concept of pruning altogether, adding every newly generated PEG-path to the priority queue. Algorithm 4 illustrates this UNAVAILING pruning strategy.

Unfortunately, this approach has the critical ruination of not progressing beyond the first conservative region boundary edge. The following claim and the accompanying discussion illustrate why this occurs.

Theorem 3. *If* PRUNABLE *is equivalent to* UNAVAILING, *then the forward search will never progress beyond the first encountered conservative region boundary edge.*

Proof. Suppose we have the following: a forward search which begins in PEG-node v at point p. PEG-node v corresponds to a convex conservative region with $k \geq 3$ sides. Without loss of generality, we say that the nearest point on edge e_1 is a distance of 1 away from point p and all points on edges $e_2, \ldots, e_k$ are a distance of at least $1+\epsilon$ away from p where $\epsilon > 0$, as seen in Figure 18. When the single element PEG-path v is expanded, there are k new PEG-paths added to the priority queue. The next PEG-path to appear at the front of the priority queue will be (v, u), where u is a PEG-node reached by travelling from v and crossing the critical boundary associated with e_1. This PEG-path has a cost of 1. Like node v, node u corresponds to a convex conservative region with j sides where $j \geq 3$. We already know that the cost of getting to node u via edge e_1 is 1. The cost to reach any other edge will be some positive value greater than 1. So, during the expansion phase, a three element PEG-path (v, u, w) will be generated. This PEG-path has a cost of 1, which is smaller than all the preexisting two-element PEG-paths and the newly generated three-element PEG-paths. This PEG-path is unique because the conservative region that corresponds to v is the same conservative region that corresponds to w. At each successive iteration, the PEG-path that appears at the top of the priority queue will reside either in the conservative region corresponding to v (odd length paths) or the conservative region corresponding to u (even length paths) and will have a cost of 1. No PEG-path with a ToS of length longer than 1 will ever be expanded. □

This unsatisfying result demonstrates the necessity of non-vacuous pruning. Without discarding at least some PEG-path, the algorithm will never terminate.

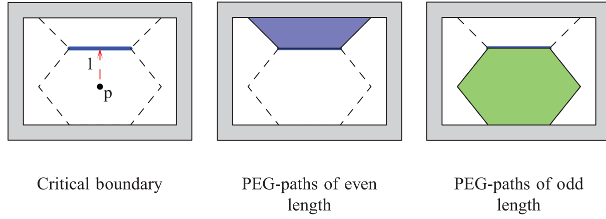


Fig. 18. The scenario that occurs when the UNAVAILING pruning strategy is used. Initially the pursuer must travel to the closest critical boundary edge. Then for PEG-paths of even length, the pursuer will be in the blue conservative region. For PEG-paths of odd length, the pursuer will be in the green conservative region.

Algorithm 4 UNAVAILING

Input: a PEG-path $\hat{v} = (v_1 \dots v_n)$

1: **function** PRUNABLE($\hat{v}$)
2: **return** false
3: **end function**

Algorithm 5 CYCLE-FREE PRUNING

Input: a PEG-path $\hat{v} = (v_1, \dots, v_n)$

1: **function** PRUNABLE(V)
2: **for each** $v_i \in \hat{v}, \quad i \neq n$ **do**
3: **if** $\mathrm{cr}(v_i) = \mathrm{cr}(v_n)$ and $\mathrm{label}(v_i) = \mathrm{label}(v_n)$ **then**
4: **return** true
5: **end if**
6: **end for**
7: **return** false
8: **end function**

8.2. Cycle-free pruning

One simple approach that can eliminate the oscillations that occur in the proof of Theorem 3 is to prune paths in which a PEG-node appears twice. Algorithm 5 illustrates this idea.

Prior to adding each new PEG-path to the priority queue, the path is checked to ensure that the newly appended node is unique in that path. Because each PEG-path is formed by appending a single PEG-node to an existing path that has previously passed the CYCLE-FREE test, we need only check for duplicates with the new final PEG-node, rather than between all pairs.

The following table shows some illustrative examples for how the CYCLE-FREE pruning would process a PEG-path in a hypothetical environment. Recall that a PEG-path is represented by a sequence of PEG-nodes, within which a single PEG-node is represented as a conservative region (shown below by their integer labels) and shadow label pair.

Note that a more restrictive approach in which we consider pruning PEG-paths that revisit the same conservative region (rather than the same PEG-node) would be incorrect. There are many environments for which backtracking to a previously visited portion of the environment is necessary

PEG-path	Pruned?	Reasoning
(7,111), (8,11), (4,10), (8,10)	No	The most recent PEG-node (8,10) does not appear previously in the PEG-path.
(7,111), (8,11), (3,101), (8,11)	Yes	The most recent PEG-node (8,11) appears previously in the PEG-path.

to catch the evader. For instance, if the pursuer were to start in the central region of Figure 9, a solution strategy would require the pursuer to revisit this region during its search.

8.3. Pruning via shadow label dominance

This pruning technique can be extended by not only checking for cycles, but also, checking to make sure a path does not visit a PEG-node whose shadow label is dominated by a PEG-node that appears earlier in the path, in the sense of reaching the same conservative region having cleared only a subset of the shadows cleared by the other PEG-node.

The general idea is that if by following a path that corresponds to the current PEG-path, pursuers "loses" information, then that path has a suffix that can be safely discarded, since the original path contains at least as much information as the new, longer path.

The following definitions formalize this idea, introducing a dominance relation for shadow labels that can be used to prune suboptimal paths. Consider two shadow labels $L = (l_1 l_2 \cdots l_k)$ and $L' = \left(l'_1 l'_2 \cdots l'_k\right)$ for the same conservative region.

Definition 11. A shadow label L *dominates* another shadow label L', denoted $L \underline{\gg} L'$, if

$$\forall i \quad 1 \leq i \leq k \quad : \quad l_i \leq l'_i$$

A shadow label L *strictly dominates* another shadow label L', denoted $L \gg L'$, if $L \underline{\gg} L'$ and $L \neq L'$.

Informally, L dominates L' if for every shadow that is cleared in L', the corresponding shadow in L is also cleared. The intuition is that L provides at least as much information as L', and can potentially contain more information in the case where $l_i = 0$ and $l'_i = 1$.

The value of this relation is that, if $L \underline{\gg} L'$, then any pursuer path that captures the evader starting from L' will also capture the evader starting from L. A pruning strategy that uses this idea to prune suboptimal partial solutions appears in Algorithm 6. Note that this approach subsumes the simpler CYCLE-FREE pruning operation from Section 8.2.

The following table shows some illustrative examples for how the REGRESSION pruning would process a PEG-path. As in the previous table, each PEG-path is a sequence of PEG-nodes, each of which is represented as a conservative region identifier along with shadow labels.

Algorithm 6 Regression Pruning

Input: a PEG-path $\hat{v} = (v_1, \ldots, v_n)$

```
1: function PRUNABLE(v̂)
2:   for each v_i ∈ v̂,  i ≠ n do
3:     if cr(v_i) = cr(v_n) and label(v_i) ≥≥ label(v_n) then
4:       return true
5:     end if
6:   end for
7:   return false
8: end function
```

PEG-path	Pruned?	Reasoning
(7,111), (8,11), (4,10), (8,10)	No	The most recent PEG-node (8,10) is not dominated by another PEG-node that appears earlier in the sequence.
(7,111), (8,11), (3,101), (8,11)	Yes	The most recent PEG-node (8,11) is strictly dominated by PEG-node (8,11) which appears earlier in the sequence. In this scenario, we have encountered a cycle which is encapsulated in our definition of strict dominance.
(9,1011), (2,110), (13,1001), (7,111), (9,1101)	No	The most recent PEG-node (9, 1101) is not strictly dominated by another PEG-node that appears earlier in the sequence.
(9,1011), (2,110), (6,11), (11,111), (9,1111)	Yes	The most recent PEG-node (9,1111) is strictly dominated by PEG-node (9,1011) which appears earlier in the sequence. By having the pursuer visit the PEG-nodes between (9,1011) and (9,1111) we are less informed about the evader's potential location.

8.4. Pruning via path dominance

The previous two pruning operations have been based on identifying redundancies within a single PEG-path. We now consider pruning operations that are based on comparisons between pairs of paths. We introduce a notion of PEG-path dominance that we use to construct more aggressive pruning strategies.

The next definition introduces a relation between pairs of PEG-paths that allows us to establish that one of those paths can be pruned.

Definition 12. A PEG-path $\hat{v} = (v_1, \ldots, v_n)$ *dominates* another PEG-path $\hat{u} = (u_1, \ldots, u_m)$ if

(i) $\mathrm{cr}(v_n) = \mathrm{cr}(u_m)$;
(ii) $\mathrm{label}(v_n) \gneqq \mathrm{label}(u_m)$; and
(iii) for any successively adjacent sequence of conservative regions $\hat{a} = (a_1, \ldots, a_k)$, with $a_1 = \mathrm{cr}(v_n) = \mathrm{cr}(u_m)$, we have

$$\mathrm{length}(\,\mathrm{ToS}(\,\mathrm{cr}(v_1), \ldots, \mathrm{cr}(v_{n-1}), a_1, \ldots, a_k))$$
$$\le \mathrm{length}(\,\mathrm{ToS}(\,\mathrm{cr}(u_1), \ldots, \mathrm{cr}(u_{m-1}), a_1, \ldots, a_k))$$

The intuition is to describe conditions under which $\hat{v}$ can be treated as a "replacement" for $\hat{u}$, without any chance of increasing the length of the final solution strategy. If $\hat{v}$ dominates $\hat{u}$, it is harmless to discard $\hat{u}$. The following theorem makes this idea more precise.

Theorem 4. *Consider any two PEG-paths $\hat{v} = (v_1, \ldots, v_n)$ and $\hat{u} = (u_1, \ldots, u_m)$, for which $\hat{v}$ dominates $\hat{u}$. Then for any solution strategy that starts by passing through $\hat{u}$, there exists another solution strategy of equal or lesser length that starts by passing though $\hat{v}$.*

Proof. Let $\mathrm{ToS}(u_1, \ldots, u_{m-1}, w_1, \ldots, w_k)$ denote a solution strategy, in which $w_1, \ldots, w_k$ is the unique suffix PEG-path reached after passing through $\hat{u}$. Consider the path $\mathrm{ToS}(\,\mathrm{cr}(v_1), \ldots, \mathrm{cr}(v_{n-1}), \mathrm{cr}(\,w_1), \ldots, \mathrm{cr}(\,w_k))$. Conditions (i) and (ii) of Definition 12 ensure that this path is a solution strategy. Condition (iii) ensures that is path is of equal length or shorter. □

To leverage this idea in the context of a pruning operation for Algorithm 3, we maintain, for each conservative region r, a list $nd(\,r)$ of PEG-paths that reach this region, for which no other dominating path is known. When a new PEG-path $\hat{v} = (v_1, \ldots, v_n)$ is generated, there are two cases.

1. If $nd(\,\mathrm{cr}(v_n))$ contains a PEG-path $\hat{u}$ that dominates $\hat{v}$, we return true to prune $\hat{v}$.
2. If no PEG-path in $nd(\,\mathrm{cr}(v_n))$ dominates $\hat{v}$, then we (a) remove from $nd(\,\mathrm{cr}(v_n))$ any $\hat{u}$ dominated by $\hat{v}$, and (b) insert $\hat{v}$ into $nd(\,\mathrm{cr}(v_n))$. The pruning operation then returns false.

Unfortunately, this approach relies on an ability to test for dominance between pairs of PEG-paths. Because condition (iii) of Definition 12 must account for *any* suffix of successively adjacent conservative regions, it is not immediately clear how to perform this test.

Given a pair of PEG-paths $\hat{v} = (v_1, \ldots, v_n)$ and $\hat{u} = (u_1, \ldots, u_m)$, that satisfy conditions (i) and (ii) of Definition 12, let s_v and s_u denote the final conservative region boundary edges crossed by $\hat{v}$ and $\hat{u}$ respectively. Then, a conservative condition for verifying that condition (iii) of Definition 12 is satisfied is to show that

$$\mathrm{length}\left(\mathrm{ToS}(\,\hat{v})\right) + \max_{(p,q)\in s_v \times s_u} ||p - q|| \le \mathrm{length}\left(\mathrm{ToS}(\,\hat{u})\right)$$

See Figure 19. This inequality implies condition (iii), because it tests for worst-case behavior, in which the ToS for $\hat{v}$ arrives at s_v at a maximal distance from s_u which

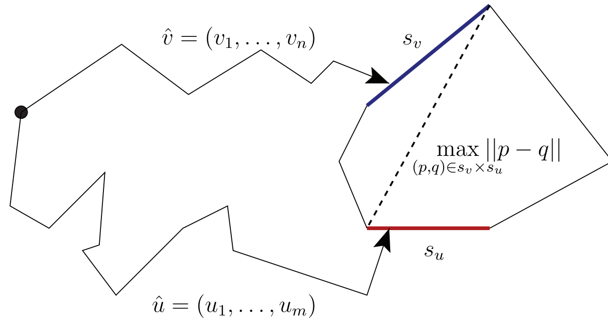


Fig. 19. An illustration of the QUADRILATERAL PRUNING test performed by Algorithm 7 which computes the diameter of the quadrilateral formed by segments s_v and s_u (the final conservative region boundaries crossed by $\hat{v}$ and $\hat{u}$ respectively).

Algorithm 7 QUADRILATERAL PRUNING

Input: a PEG-path $\hat{v} = (v_1, \ldots, v_n)$
Data: a data structure *nd* that maintains a list of non-dominated PEG-paths that reach a given conservative region

```
 1: function PRUNABLE(v̂)
 2:   for each û ∈ nd(cr(v_n)) do                    ▷ û =(u_1, ..., u_m)
 3:     s_v ← BOUNDARYSEGMENT(cr(v_{n−1}), cr(v_n))
 4:     s_u ← BOUNDARYSEGMENT(cr(u_{m−1}), cr(u_m))
 5:     length_U ← TOUROFSEGMENTS(û)
 6:     dist ← max_{(p,q)∈s_v×s_u} ||p − q||
 7:     length_V ← TOUROFSEGMENTS(v̂)
 8:     if label(u_m) ≫ label(v_n) then              ▷ û dominates v̂
 9:       if length_U + dist < length_V then         ▷ û is always preferable
10:         return true
11:       end if
12:     else if label(v_n) ≫ label(u_m) then         ▷ v̂ strictly dominates û
13:       if length_V + dist < length_U then         ▷ v̂ is always preferable
14:         nd.remove(û)
                      ▷ Remove û from the "non-dominated" list
15:       end if
16:     end if
17:   end for
18:   return false
19: end function
```

is the entry point for $\hat{u}$ into the conservative region. Note that, aside from the two ToS queries (whose values would already have been computed and cached), this inequality can be evaluated in constant time.

A pruning strategy based on this approach appears in Algorithm 7.

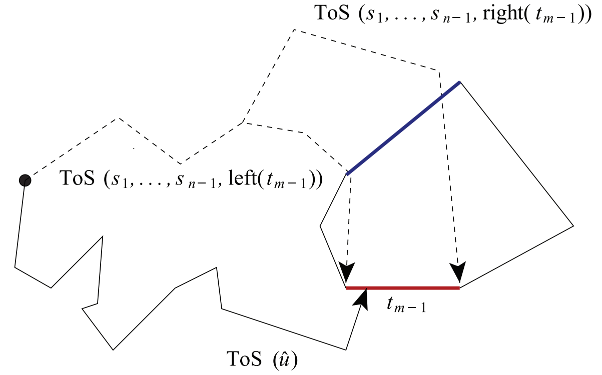


Fig. 20. An illustration of the ENDPOINT PRUNING test performed by Algorithm 8 which requires the ToSes through $\hat{v}$ that terminate on the left and right endpoints of segment t_{m-1} to outperform the ToSes through $\hat{u}$ that terminate on t_{m-1}.

8.5. Endpoint pruning

A slightly more aggressive condition for stating that condition (iii) of Definition 12 is satisfied for PEG-paths $\hat{v} = (v_1, \ldots, v_n)$ and $\hat{u} = (u_1, \ldots, u_m)$ is to show that, for any successively adjacent sequence of conservative regions $\hat{a} = (a_1, \ldots, a_k)$, with $a_1 = \text{cr}(v_n) = \text{cr}(u_m)$, we have

$$\text{length}(\text{ ToS}\left(\text{cr}(v_1), \ldots, \text{cr}(v_n), \text{cr}(u_{m-1}), a_1, \ldots, a_k\right)) \leq \text{length}(\text{ ToS}\left(\text{cr}(u_1), \ldots, \text{cr}(u_{m-1}), a_1, \ldots, a_k\right))$$

One sufficient condition for testing this property is to let $s_1, \ldots, s_{n-1}$ denote the conservative region boundary segments crossed by $\hat{v}$, and let t_{m-1} denote the final such segment crossed by $\hat{u}$. We then check whether

$$\text{length}\left(\text{ToS}(s_1, \ldots, s_{n-1}, \text{left}(t_{m-1})\right) \leq \text{length}\left(\text{ToS}(\hat{u})\right)$$

and

$$\text{length}\left(\text{ToS}(s_1, \ldots, s_{n-1}, \text{right}(t_{m-1})\right) \leq \text{length}\left(\text{ToS}(\hat{u})\right)$$

in which left side of each inequality uses degenerate segment, that is, a single point, as the final step.

The intuition is that by evaluating the distance at each of the endpoints of t_{m-1}, we can find worst-case distance that any optimal tour would travel through $\hat{v}$ followed by t_{m-1}. This occurs because the furthest distance between a point and a line segment occurs at one of the endpoints. An illustration of this process appears in Figure 20.

A pruning operation, based on this idea along with the non-dominated lists introduced in Section 8.4, appears in Algorithm 8. Compared to Algorithm 7, which takes a small constant amount of time, this approach must solve two new ToS queries each time. Our evaluation in Section 9 demonstrates that the improved pruning capability outweighs the overhead of associated with computing the additional tours.

Algorithm 8 ENDPOINT PRUNING

Input: a PEG-path $\hat{v} = (v_1, \ldots, v_n)$
Data: a data structure *nd* that maintains a list of non-dominated PEG-paths that reach a given conservative region

```
1:  function PRUNABLE(v̂)
2:    for each û ∈ nd(cr(v_n)) do                                      ▷ û = (u_1, ..., u_m)
3:      s_1 ... s_{n-1} ← BOUNDARYSEGMENTS(cr(v_1), ..., cr(v_n))
4:      t_1 ... t_{m-1} ← BOUNDARYSEGMENTS(cr(u_1), ..., cr(u_m))
5:      if label(u_m) ≫ label(v_n) then                                ▷ û dominates v̂
6:        length_{U_L} ← TOUROFSEGMENTS(t_1, ..., t_{m-1}, left(s_{n-1}))
7:        length_{U_R} ← TOUROFSEGMENTS(t_1, ..., t_{m-1}, right(s_{n-1}))
8:        length_V ← TOUROFSEGMENTS(s_1, ..., s_{n-1})
9:
10:       ▷ Path through û is always preferable
11:       if ((length_{u_p1} < length_V) and (length_{u_p2} < length_V)) then
12:         return true
13:       end if
14:     else if label(v_n) ≫ label(u_m) then                           ▷ v̂ strictly dominates û
15:       length_{V_L} ← TOUROFSEGMENTS(s_1, ..., s_{n-1}, left(t_{m-1}))
16:       length_{V_R} ← TOUROFSEGMENTS(s_1, ..., s_{n-1}, right(t_{m-1}))
17:       length_U ← TOUROFSEGMENTS(t_1, ..., t_{m-1})
18:
19:       ▷ Path through v̂ is always preferable
20:       if ((length_{V_L} < length_U) and (length_{V_R} < length_U)) then
21:         nd.remove(û)
                                   ▷ Remove û from the "non-dominated" list
22:       end if
23:     end if
24:   end for
25:   return false
26: end function
```

8.6. Lookahead pruning

A final, more aggressive pruning strategy can be formed by considering the conservative region boundary segment crossed to *leave* the final conservative region.

Given a pair of PEG-paths $\hat{v} = (v_1, \ldots, v_n)$ and $\hat{u} = (u_1, \ldots, u_m)$, let $s_1, \ldots, s_{n-1}$ denote the conservative region boundary segments crossed by $\hat{v}$, and let $t_1, \ldots, t_{m-1}$ denote the segments crossed by $\hat{u}$.

Consider the final conservative region $r = \mathrm{cr}(v_n) = \mathrm{cr}(u_m)$. Any extension of these paths must exit region r on its next step. Let s denote the segment, which must be a boundary segment of r, on which this occurs.

Then if, for any s on the boundary of r, we have

$$\text{length}\left(\text{ToS}(s_1, \ldots, s_{n-1}, \text{left}(s)\right) \leq \text{length}\left(\text{ToS}(t_1, \ldots, t_{m-1}, \text{left}(s)\right)$$

and

$$\text{length}\left(\text{ToS}(s_1, \ldots, s_{n-1}, \text{right}(s)\right) \leq \text{length}\left(\text{ToS}(t_1, \ldots, t_{m-1}, \text{right}(s)\right)$$

then for any successively adjacent sequence of conservative regions $(a_1, \ldots, a_k)$ with $a_1 = r$, we have

$$\text{length}(\text{ToS}(\text{cr}(v_1), \ldots, \text{cr}(v_{n-1}), a_1, \ldots, a_k)) \leq \text{length}(\text{ToS}(\text{cr}(u_1), \ldots, \text{cr}(u_{m-1}), a_1, \ldots, a_k))$$

The intuition is that by performing the ToS subroutine on every potential future segment s, we can test to see if sequence $\hat{v}$ will always be preferred to sequence $\hat{u}$. Like the reasoning behind the Endpoint Pruning, we check both endpoints because the farthest distance between a point and a line segment occurs at one of the endpoints. This effectively means that sequence $\hat{v}$ must have a shorter tour to both endpoints of any potential future segment s for it to dominate sequence $\hat{u}$. An implementation of this strategy appears in Algorithm 9.

Algorithm 9 LOOKAHEAD PRUNING

Input: a PEG-path $\hat{v} = (v_1, \ldots, v_n)$
Data: a data structure *nd* that maintains a list of non-dominated PEG-paths that reach a given conservative region

1: **function** PRUNABLE($\hat{v}$)
2: **for each** $\hat{u} \in nd(\mathrm{cr}(v_n))$ **do** ▷ $\hat{u} = (u_1, \ldots, u_m)$
3: $s_1, \ldots, s_{n-1} \leftarrow$ BOUNDARYSEGMENTS$(\mathrm{cr}(v_1), \ldots, \mathrm{cr}(v_n))$
4: $t_1, \ldots, t_{m-1} \leftarrow$ BOUNDARYSEGMENTS$(\mathrm{cr}(u_1), \ldots, \mathrm{cr}(u_m))$
5: **if** $\mathrm{label}(u_m) \gg \mathrm{label}(s_n)$ **then** ▷ $\hat{u}$ dominates $\hat{v}$
6: **for each** $seg \in$ CRBOUNDARYSEGMENTS$(\mathrm{cr}(v_n))$ **do**
7: $\mathrm{length}_{U_L} \leftarrow$ TOUROFSEGMENTS$(t_1, \ldots, t_{m-1}, \mathrm{left}(seg))$
8: $\mathrm{length}_{U_R} \leftarrow$ TOUROFSEGMENTS$(t_1, \ldots, t_{m-1}, \mathrm{right}(seg))$
9: $\mathrm{length}_{V_L} \leftarrow$ TOUROFSEGMENTS$(s_1, \ldots, s_{n-1}, \mathrm{left}(seg))$
10: $\mathrm{length}_{V_R} \leftarrow$ TOUROFSEGMENTS$(s_1, \ldots, s_{n-1}, \mathrm{right}(seg))$
11: ▷ Path through $\hat{u}$ is not preferable for all potential segments
12: **if** $((\mathrm{length}_{U_L} > \mathrm{length}_{V_L})$ **or** $(\mathrm{length}_{U_R} > \mathrm{length}_{V_R}))$ **then**
13: **Continue to next** $\hat{u}$
14: **end if**
15: **end for**
16: ▷ Path through $\hat{u}$ is always preferable
17: **return** true
18: **else if** $\mathrm{label}(v_n) \gg \mathrm{label}(u_m)$ **then** ▷ $\hat{v}$ strictly dominates $\hat{u}$
19: **for each** $seg \in$ CRBOUNDARYSEGMENTS$(\mathrm{cr}(v_n))$ **do**
20: $\mathrm{length}_{U_L} \leftarrow$ TOUROFSEGMENTS$(t_1, \ldots, t_{m-1}, \mathrm{left}(seg))$
21: $\mathrm{length}_{U_R} \leftarrow$ TOUROFSEGMENTS$(t_1, \ldots, t_{m-1}, \mathrm{right}(seg))$
22: $\mathrm{length}_{V_L} \leftarrow$ TOUROFSEGMENTS$(s_1, \ldots, s_{n-1}, \mathrm{left}(seg))$
23: $\mathrm{length}_{V_R} \leftarrow$ TOUROFSEGMENTS$(s_1, \ldots, s_{n-1}, \mathrm{right}(seg))$
24: ▷ Path through $\hat{v}$ is not preferable for all potential segments
25: **if** $((\mathrm{length}_{V_L} > \mathrm{length}_{U_L})$ **or** $(\mathrm{length}_{V_R} > \mathrm{length}_{U_R}))$ **then**
26: **Continue to next** $\hat{u}$
27: **end if**
28: **end for**
29: ▷ Path through $\hat{v}$ is always preferable
30: *nd*.remove($\hat{u}$) ▷ Remove $\hat{u}$ from the "non-dominated" list
31: **end if**
32: **end for**
33: **return** false
34: **end function**

9. Evaluation

We have implemented all the algorithms described in this paper. In this section, we provide simulated results to evaluate the effectiveness of our approach. Our implementation uses C++, and timing results are for a single core of an 2.4Ghz Intel i7 processor.

We used our implementation to solve the visibility-based pursuit-evasion problem in five different environments.

1. Figure 21. This environment has 57 conservative regions, with a total of 21,806 PEG-nodes. The number of shadows per conservative region is at most 11. The PEG was constructed in 1.112 seconds.
2. Figure 22. This environment has 125 conservative regions, with a total of 35,530 PEG-nodes. The number of shadows per conservative region is at most 10. The PEG was constructed in 2.840 seconds.

3. Figure 23. This environment has 213 conservative regions, with a total of 24, 620 PEG-nodes. The number of shadows per conservative region is at most 11. The PEG was constructed in 43.396 seconds.
4. Figure 24. This environment has 491 conservative regions, with a total of 56, 888 PEG-nodes. The number of shadows per conservative region is at most 11. The PEG was constructed in 1011.800 seconds.
5. Figure 25. This environment has 282 conservative regions, with a total of 69, 806 PEG-nodes. The number of shadows per conservative region is at most 11. The PEG was constructed in 65.636 seconds.

We attempted to solve each environment with several different algorithms.

1. The original GL^3M algorithm.
2. A modified version of GL^3M, in which the generated path undergoes a post-processing step, a ToS, to find the shortest path that follows the same PEG-path. This approach is intended to provide a more fair comparison to our approach.
3. Five variations of our algorithm, one for each of the pruning operations described in Algorithms 5 to 9.

The paths generated by these approaches are shown in Figures 21 to 25.

Observe in Figures 21, 23, and 24 that the optimal paths require the pursuer to change direction at a point that is neither an environment vertex nor a vertex of the decomposition. These direction changes arise from bounce events in the shortest path maps.

We also measured the run time of each algorithm for each of these environments. Each trial was limited to 15 minutes of CPU time and 8Gb of RAM. Trials that exceeded these limits were considered failures. Figure 26 shows the results. For those trials unable to generate a solution strategy within the constraints described above, we measured the ToS length of the longest path expanded. Figure 27 shows these results as a percentage of the optimal path returned using the LOOKAHEAD pruning. Since the search would terminate upon expanding the actual optimal path, one can think of these fractions as a proxy for progress made by the algorithm. Finally, for those trials that generated a solution strategy, we display the length of the returned strategy (as a percentage of the optimal strategy length) in Figure 28.

For the non-trivial environments that appear in Figures 21 to 25, several of the pruning strategies were unable to generate a solution strategy. The CYCLE-FREE pruning technique from Algorithm 5 exceeded the memory allotment of 8Gb. The other pruning strategies that were unable to generate a solution; REGRESSION, QUADRILATERAL, and ENDPOINT; all failed due to the time constraint. The results in Figure 27 suggests that there is some tradeoff between employing computationally expensive pruning strategies and the progress made by the search. In general, we have seen that the benefits of using one of the more rigorous pruning strategies to disregard suboptimal partial strategies outweighs the overhead they incur. Only the environments that appear in Figures 21 and 22 have instances where the naiver strategies outperform their more complicated brethren.

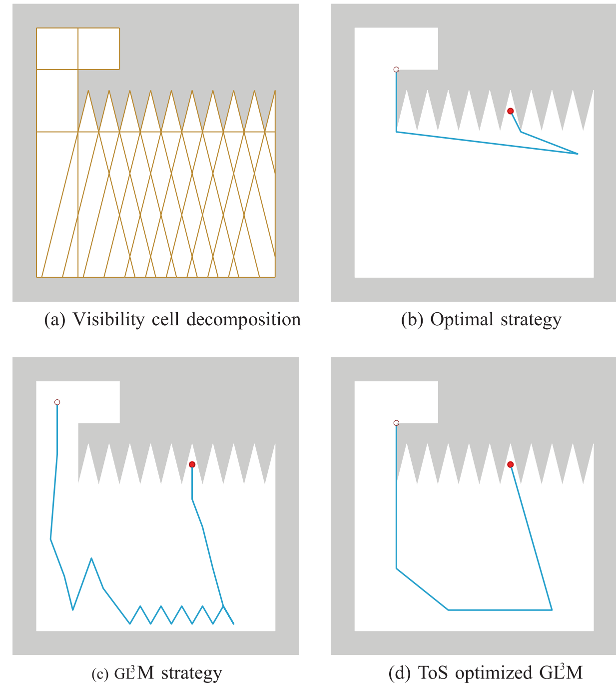
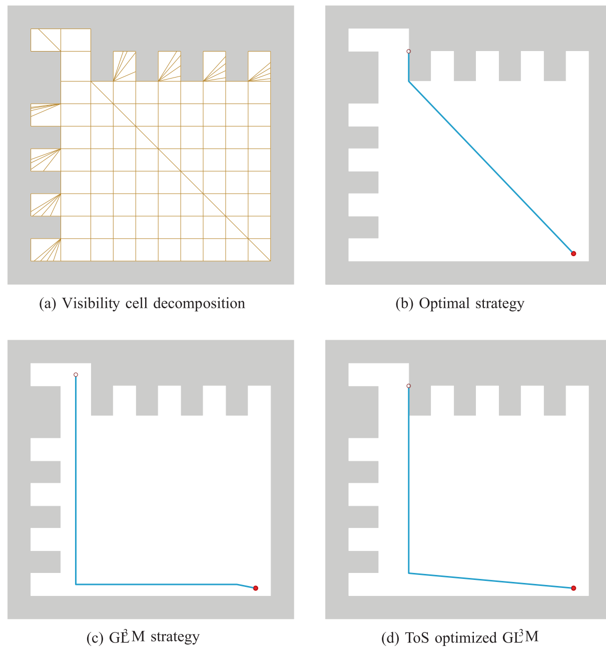


Fig. 21. An environment (21a) where the optimal pursuer strategy (21b) returned by our algorithm looks vastly different from both the original GL^3M strategy (21c) and the GL^3M strategy optimized using a ToS (21d).

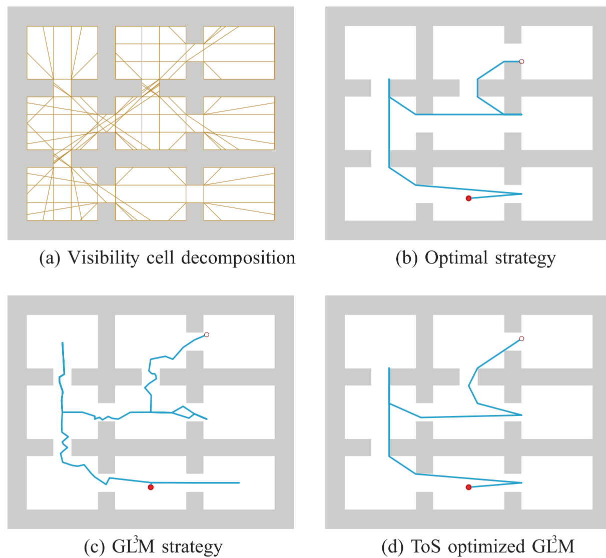
The results of Figure 28 clearly indicate that our algorithm that uses the LOOKAHEAD pruning strategy returns solution strategies that are far superior to that of the GL^3M algorithm and the GL^3M algorithm that uses the ToS subroutine as a post-processing improvement. Figures 21 and 22 show the drastic improvement in tour length that occurs as a result of using our algorithm. Even the more typical office-like environments of Figures 23 and 24 experience an improvement by employing our algorithm. The environment in Figure 25 is yet another example where our algorithm significantly outperforms the GL^3M algorithm. This environment is of historical interest because it is a known environment presented in the Guibas, Latombe, LaValle, Lin, and Motwani work that requires the pursuer to revisit the topmost portion of the environment a linear number of times (based on the number of forked corridors at the bottom) due to recontamination. The solution generated by our algorithm is a marked improvement over the GL^3M and GL^3M with ToS algorithms which have the pursuer travel down the left-side of the environment prior to visiting each pair of forked corridors.



(a) Visibility cell decomposition (b) Optimal strategy

(c) GL^{3}M strategy (d) ToS optimized GL^{3}M

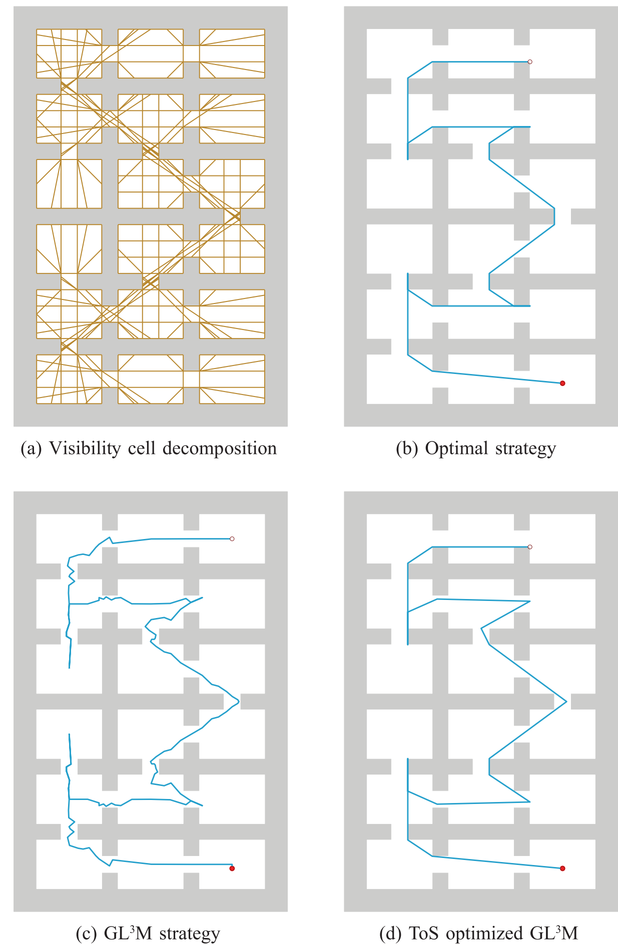
Fig. 22. An environment (22a) where the optimal pursuer strategy (22b) returned by our algorithm looks vastly different from both the original GL^{3}M strategy (22c) and the GL^{3}M strategy optimized using a ToS (21d).



(a) Visibility cell decomposition (b) Optimal strategy

(c) GL^{3}M strategy (d) ToS optimized GL^{3}M

Fig. 23. An environment (23a) where the optimal pursuer strategy (22b) returned by our algorithm looks fairly similar to both the original GL^{3}M strategy (22c) and the GL^{3}M strategy optimized using a ToS (22d).



(a) Visibility cell decomposition (b) Optimal strategy

(c) GL^{3}M strategy (d) ToS optimized GL^{3}M

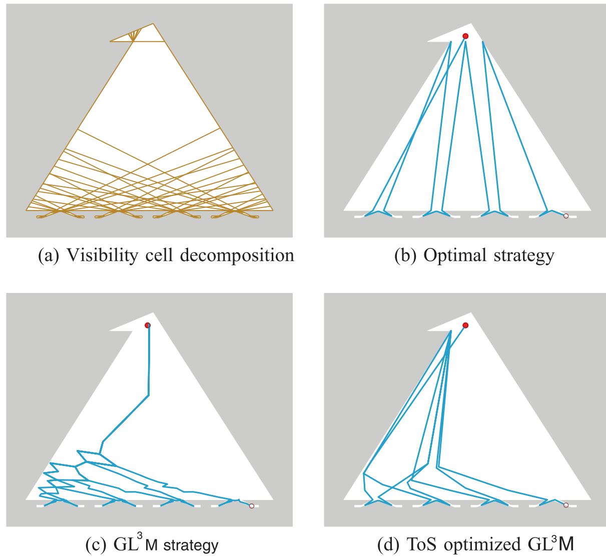
Fig. 24. An environment (24a) where the optimal pursuer strategy (24b) returned by our algorithm looks fairly similar to both the original GL^{3}M strategy (24c) and the GL^{3}M strategy optimized using a ToS (24d).

10. Conclusion

This result improves upon the known result of Guibas, Latombe, LaValle, Lin, and Motwani that returns a feasible solution strategy for a single pursuer in a simply-connected polygonal environment by solving for the minimal length solution strategy. The remainder of this section considers two orthogonal threads of future work.

First, similar to how this work extends the GL^{3}M algorithm by computing optimal solutions, a potential avenue for future work would be to perform a similar optimization on other pursuit-evasion problems. Recall from Section 2.3.1 the variety of single pursuer visibility-based pursuit-evasion problems that differ based on the pursuer's sensing and motion capabilities.

Lastly, due to the exponential nature of the PEG, it is reasonable to expect problem instances where the number of candidate sequences to consider will begin to make the problem computationally intractable. Under these circumstances, it would be beneficial to investigate how an approximation algorithm (Vazirani, 2001; Williamson and Shmoys, 2011) could be harnessed to provide some semblance of performance guarantees. The general idea behind an approximation algorithm is that it produces solutions that remain within some constant factor of the optimal solution while typically requiring reduced computation expense. The potential performance gains are enough of an incentive to at least motivate discussion of the applicability of approximation algorithms in solving visibility-based pursuit-evasion problems.



(a) Visibility cell decomposition (b) Optimal strategy

(c) GL^{3}M strategy (d) ToS optimized GL^{3}M

Fig. 25. An environment (25a) where the optimal pursuer strategy (25b) returned by our algorithm looks fairly similar to both the original GL^{3}M strategy (25c) and the GL^{3}M strategy optimized using a ToS (25d).

Env	GL^{3}M	GL^{3}M-ToS	C	R	Q	E	L
Figure 21	0.00045	0.00046	MLE	DNF	DNF	DNF	8.320
Figure 22	0.00033	0.00038	MLE	DNF	DNF	DNF	4.804
Figure 23	0.00226	0.00249	MLE	DNF	DNF	DNF	8.784
Figure 24	0.00540	0.00724	MLE	DNF	DNF	DNF	32.520
Figure 25	0.00699	0.00739	MLE	DNF	DNF	DNF	611.696

Fig. 26. Timing results, in seconds, for the environments in Figures 21 to 25. 'MLE' indicates 'memory limit exceeded'. 'DNF' indicates that the algorithm 'did not finish' within the time allotted. All pruning strategies are denoted by the first letter in the pruning strategies name. Cycle-free=C, Regression=R, Quadrilateral=Q, Endpoint=E, Lookahead=L.

Env	GL^{3}M	GL^{3}M-ToS	C	R	Q	E	L
Figure 21	–	–	54.78%	50.70%	53.81%	71.28%	–
Figure 22	–	–	60.77%	51.58%	57.02%	55.32%	–
Figure 23	–	–	20.20%	21.52%	34.41%	41.17%	–
Figure 24	–	–	18.87%	18.31%	25.71%	27.11%	–
Figure 25	–	–	12.31%	12.45%	14.47%	14.23%	–

Fig. 27. The ToS length for the longest expanded PEG-path as a percentage of the length of the optimal solution strategy. Values for successful runs are omitted. All pruning strategies are denoted by the first letter in the pruning strategies name. Cycle-free=C, Regression=R, Quadrilateral=Q, Endpoint=E, Lookahead=L.

Acknowledgement

We are grateful to William Edwards for assistance in preparing this document.

Funding

This work was supported by NSF (IIS-0953503).

Env	GL^{3}M	GL^{3}M-ToS	C	R	Q	E	L
Figure 21	217.53%	141.84%	–	–	–	–	100.00%
Figure 22	145.28%	131.44%	–	–	–	–	100.00%
Figure 23	152.06%	106.69%	–	–	–	–	100.00%
Figure 24	136.91%	105.46%	–	–	–	–	100.00%
Figure 25	175.98%	121.48%	–	–	–	–	100.00%

Fig. 28. The ToS length for successfully computed paths. All pruning strategies are denoted by the first letter in the pruning strategies name. Cycle-free=C, Regression=R, Quadrilateral=Q, Endpoint=E, Lookahead=L.

Notes

1. The objective of the watchman route problem 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.
2. Bug algorithms (Lumelsky and Stepanov, 1987) assume only local knowledge of the environment and a global goal. The behaviors typically available to a "bug" include wall following and straight line motions toward the goal. Most instances of bug algorithms lack a map and the ability to construct a map and may account for imperfect navigation.
3. Note that an individual shadow may change between cleared and contaminated many times; a shadow whose label changes from cleared to contaminated is said to be *recontaminated*. There are problem instances that require the pursuer to clear a recontaminated shadow (Guibas et al., 1999). Our algorithm correctly handles the issue of recontamination when generating plans for the pursuer.
4. In this context, general position assumes that no three environment vertices are colinear.

References

Alspach B (2004) Searching and sweeping graphs: A brief survey. *Matematiche* 59: 5–37.

Baxter J, Burke E, Garibaldi J, et al. (2007) Multi-robot search and rescue: A potential field based approach. In: Mukhopadhyay S and Gupta G (eds) *Autonomous Robots and Agents*, *Studies in Computational Intelligence*, volume 76. Heidelberg: Springer Berlin, pp. 9–16.

Bienstock D (1991) Graph searching, path-width, tree-width and related problems (A Survey). In: Roberts F, Hwang F and Monma C (eds) *Reliability Of Computer And Communication Networks, Proceedings of a DIMACS Workshop, volume 5 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, New Brunswick, New Jersey, USA, 2–4 December 1989, pp. 33–49. Providence, RI: DIMACS/AMS.

Borie R, Koenig S and Tovey C (2013) Pursuit-evasion problems. In: Gross J, Yellen J and Zhang P (eds) *Handbook of Graph Theory*, chapter 9.5, Boca Raton, FL: Chapman and Hall/CRC, pp. 1145–1165.

Calisi D, Farinelli A, Iocchi L, et al. (2007) Multi-objective exploration and search for autonomous rescue robots: Research articles. *Journal of Field Robotics* 24(8-9): 763–777.

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

Dror M, Efrat A, Lubiw A, et al. (2003) Touring a sequence of polygons. In: *Proceedings of the ACM symposium on*

theory of computing, New York, NY, USA, 9–11 June 2003, pp. 473–482. ACM Press.

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

Fomin FV and Thilikos DM (2008) An annotated bibliography on guaranteed graph searching. *Theoretical Computer Science* 399(3): 236–245.

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

Golovach P (1989) A topological invariant in pursuit problems. *Differentsial'nye Uraveniya (Differential Equations)* 25: 923–929.

Guibas LJ, Latombe JC, LaValle SM, et al. (1999) Visibility-based pursuit-evasion in a polygonal environment. *International Journal on Computational Geometry and Applications* 9(5): 471–494.

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

Isaacs R (1965) *Differential Games*. New York: Wiley.

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

Karnad N and Isler V (2009) Lion and man game in the presence of a circular obstacle. In: *Proceedings of the IEEE/RSJ international conference on intelligent robots and systems*. Piscataway, NJ, USA, 10–15 October 2009, pp. 5045–5050. IEEE.

Klein K and Suri S (2013) Capture bounds for visibility-based pursuit evasion. In: *Proceedings of the ACM symposium on computational geometry*, New York, NY, USA, 17–20 June 2013, pp. 329–338. ACM.

Kleiner A, Farinelli A, Ramchurn S, et al. (2013) Rmasbench: Benchmarking dynamic multi-agent coordination in urban search and rescue. In: *Proceedings of the international conference on autonomous agents and multiagent systems*, Richland, SC, 6–10 May 2013, pp. 1195–1196. International Foundation for Autonomous Agents and Multiagent Systems.

Kleiner A and Kolling A (2013) Guaranteed search with large teams of unmanned aerial vehicles. In: *Proceedings of the IEEE international conference on robotics and automation* 6–10 May 2013, pp. 2977–2983. IEEE.

Kolling A and Carpin S (2009) Surveillance strategies for target detection with sweep lines. In: *Proceedings of the IEEE/RSJ international conference on intelligent robots and systems*, Piscataway, NJ, USA, 10–15 October 2009, pp. 5821–5827. IEEE.

Kolling A and Carpin S (2010) Pursuit-evasion on trees by robot teams. *IEEE Transactions on Robotics* 26(1): 32–47.

LaValle SM (2006) *Planning Algorithms*. Cambridge: Cambridge University Press.

LaValle SM and Hinrichsen J (2001) Visibility-based pursuit-evasion: The case of curved environments. *IEEE Transactions on Robotics and Automation* 17(2): 196–201.

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

Lee J, Park S and Chwa K (2002) Simple algorithms for searching a polygon with flashlights. *Information Processing Letters* 81(5): 265–270.

Lumelsky VJ and Stepanov AA (1987) Path planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorthmica* 2: 403–430.

Murphy RR (2014) *Disaster Robotics*. Cambridge, MA: The MIT Press.

Noori N and Isler V (2012) Lion and man with visibility in monotone polygons. In: *Proceedings of the workshop on the algorithmic foundations of robotics*, *Springer tracts in advanced robotics*, volume 86, New York, NY, USA, 13–15 June 2012, pp. 263–278. Springer.

Noori N and Isler V (2014) Lion and man game on convex terrains. In: *Proceedings of the workshop on the algorithmic foundations of robotics, Springer tracts in advanced robotics*, volume 107, New York, NY, USA, 3–5 August 2014. pp. 443–460. Springer.

Park S, Lee J and Chwa K (2001) Visibility-based pursuit-evasion in a polygonal region by a searcher. In: *Proceedings of the international colloquium on automata, languages and programming*, New York, NY, USA, 8–12 July 2001, pp. 456–468, Springer–Verlag.

Parsons TD (1976) Pursuit-evasion in a graph. In: Alavi Y and Lick DR (eds) *Theory and Application of Graphs*. Berlin: Springer-Verlag, pp. 426–441.

Petrov NN (1982) A problem of pursuit in the absence of information on the pursued. *Differentsial'nye Uraveniya (Differential Equations)* 18: 1345–1352.

Petrov NN (1983) The cossack-robber differential game. *Differentsial'nye Uraveniya (Differential Equations)* 19: 1366–1374.

Rajko S and LaValle SM (2001) A pursuit-evasion bug algorithm. In: *Proceedings of the IEEE international conference on robotics and automation*, 21–26 May 2001, pp. 1954–1960. IEEE.

Ruiz U and Murrieta–Cid R (2013) Time-optimal motion strategies for capturing an omnidirectional evader using a differential drive robot. *IEEE Transactions on Robotics* 21(3): 1180–1196.

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

Sgall J (2001) Solution of David Gale's lion and man problem. *Theoretical Computer Science* 259(1–2): 663–670.

Stiffler NM and O'Kane JM (2012) Shortest paths for visibility-based pursuit-evasion. In: *Proceedings of the IEEE international conference on robotics and automation*, 14–18 May 2012, pp. 3997–4002. IEEE.

Stiffler NM and O'Kane JM (2014a) A complete algorithm for visibility-based pursuit-evasion with multiple pursuers. In: *Proceedings of the IEEE international conference on robotics and automation*, 31 May–5 June 2014, pp. 1660–1667. IEEE.

Stiffler NM and O'Kane JM (2014b) A sampling based algorithm for multi-robot visibility-based pursuit-evasion. In: *Proceedings of the IEEE/RSJ international conference on intelligent robots and systems*, 14–18 September 2014, pp. 1782–1789. IEEE.

Stiffler NM and O'Kane JM (2016) Pursuit-evasion with fixed beams. In: *Proceedings of the IEEE international conference on robotics and automation*. 16–21 May 2016, pp. 4251–4258. IEEE.

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

Abramovskaya TV and Petrov NN (2013) The theory of guaranteed search on graphs. *Vestnik St Petersburg University: Mathematics* 46(2): 49–75.

Tovar B and LaValle SM (2006) Visibility-based pursuit-evasion with bounded speed. *International Journal of Robotics Research* 27(11–12): 1350–1360.

Vander Hook J and Isler V (2014) Pursuit and evasion with uncertain bearing measurements. In: *Proceedings of the Candadian conference on computational geometry*. 11–13 August 2014, pp. 332–340.

Vazirani VV (2001) *Approximation Algorithms*. New York: Springer–Verlag.

Williamson DP and Shmoys DB (2011) *The Design of Approximation Algorithms*. 1st edition. New York: Cambridge University Press.

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

Zou R and Bhattacharya S (2016) Visibility-based finite-horizon target tracking game. *IEEE Robotics and Automation Letters* 1(1): 399–406.