6-30-2016

Visibility-Based Pursuit-Evasion In The Plane

Nicholas Michael Stiffler
University of South Carolina

Visibility-Based Pursuit-Evasion in the Plane

by

Nicholas Michael Stiffler

Bachelor of Science
University of South Carolina, 2009

Master of Science
University of South Carolina, 2012

Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy in

Computer Science and Engineering

College of Engineering and Computing

University of South Carolina

2016

Accepted by:

Jason M. O'Kane, Major Professor

Marco Valtorta, Committee Member

Michael N. Huhns, Committee Member

Manton M. Matthews, Committee Member

Linyuan Lu, External Committee Member

Lacy Ford, Senior Vice Provost and Dean of Graduate Studies

Dedication

“The true test of a man’s character is what he does when no one is watching.”

– John Wooden

For my siblings

Tyler, Lydia, Kyle, Jordan, and Blue

Whose eyes have always precluded my character from this exam.

Acknowledgments

This thesis is the culmination of several years of scholarship and would not have been possible without the generous contributions of many different people. My mentor, Jason O'Kane, had the most profound impact on my development as a research scientist and provided valuable advice along the way.

I also offer sincere thanks to my peers in the South Carolina Autonomous Robotics Research (SCARR) group: Mohammad Behbooei, Laura Boccanfuso, Shervin Ghasemloo, Jeremy Lewis, Fatemeh Zahra Saberifar, and Yang Song. The research group fostered a fertile environment in which I was able to do my best work. My committee members Michael Huhns, Linyuan Lu, Manton Matthews, and Marco Valtorta offered helpful guidance throughout by applying their unique expertise and perspective to my work. Ioannis Rekleitis, while not a serving member of my committee, also deserves recognition for his willingness to provide an external perspective on my work. Many other faculty and professors, including Ryan Austin, Randy Baldwin, Jenay Beer, Duncan Buell, Stephen Fenner, Srihari Nelakuditi, Barb Ulrich, and Gabriel Terejanu, have had immense influence on my growth and development as a computer scientist and as a person. Thank you for the the various contributions you each have made to my life.

More personally, I am thankful for the love and support that I have received from my family and friends during my time in graduate school. Thank you to David and Chelsea Hilmer, Matthew Crocket, Spencer Childress, Michael and Katie Reynolds, and Conway “Deuce” Harris, whose friendship during this arduous journey was pivotal in reaching the end.

Finally, it would be impossible to describe the amount of loving support and encouragemnt I'v recieved from my fiance, Lacie. Thank you for tolerating me throughout this journey.

Financial support for this work was provided by the NSF under award IIS-0953503 and DARPA under award N10AP20015.

Abstract

As technological advances further increase the amount of memory and computing power available to mobile robots, we are seeing an unprecedented explosion in the utilization of deployable robots for various tasks. The speed at which robots begin to enter various domains is largely dependent on the availability of robust and efficient algorithms that are capable of solving the complex planning problems inherent to the given domain. One such domain which is experiencing unprecedented growth in recent years requires a robot to detect and/or track a mobile agent or group of agents.

In these scenarios, there are typically two players with diametrically opposed goals. For matters of security, we have a guard and an intruder. The guard's goal is to ensure that if an intruder enters the premises they are caught in a timely manner. Analogously, the intruder wishes to evade detection for as long as possible. Search and rescue operations are often framed as a two-player game between rescuers and survivors. Though the survivors are unlikely to behave antagonistically, an agnostic model is useful for the rescuers to guarantee that the survivors are found, regardless of their movements. Both of these tasks, are at their core, *pursuit-evasion* problems.

There are many variants of the pursuit-evasion problem, the common theme amongst them is that one group of agents, the "pursuers", attempts to track members of another group, the "evaders". Geometric formulations of the pursuit-evasion problem require a pursuer(s) to systematically search an environment to locate one or more evaders ensuring that all evaders will be captured by the pursuer(s) in a finite time. The *visibility-based pursuit-evasion problem* is a geometric variant of the pursuit-evasion problem that defines a visibility-region which corresponds to the region of the environment that the pursuer(s) can actively perceive. If an evader lies within this visibility region then it is captured (detected).

This thesis contains four novel contributions that solve various visibility-based pursuit-evasion problems. The first contribution is an algorithm that computes the optimal (minimal path length) pursuer trajectory for a single pursuer. The second contribution is an algorithm that generates a joint motion strategy for multiple pursuers. Motivated by the result of the second contribution, the third result is a sampling-based algorithm for the multiple pursuer scenario. The fourth contribution is a complete algorithm that computes a trajectory for a pursuer that has a very limited sensor footprint.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

One of the ultimate goals in Robotics is the creation of an *autonomous system* that is capable of converting high-level task specifications from humans into low-level descriptions detailing how to accomplish the task. Planning algorithms are instrumental in creating autonomous systems. A planning algorithm *autonomously* decides the sequence of actions necessary to perform a task given an initial configuration, a collection of goal configurations, and a collection of sensors. This may appear to be a relatively straightforward process, but challenges such as effectively modelling the planning problem and designing and implementing efficient algorithms complicate the process.

The *motion planning* problem is a refinement of the planning problem. At the highest level, the *motion planning* problem asks the following question; "How can a robot decide what motions to perform in order to achieve its goal while operating in the physical world?" In the context of Robotics, the motion planning problem appears in such problems as: navigation, coverage, localization, manipulation, and pursuit-evasion.

In this thesis we focus on the pursuit-evasion problem. Although there are many variants of the pursuit-evasion problem, the common theme amongst them is that one group of agents, the *pursuers*, attempts to systematically locate the members of another group, the *evaders*. Pursuit-evasion problems are of particular interest because surveillance, evasion/detection, and search and rescue (SAR) are, at their heart, pursuit-evasion problems. The following scenarios demonstrate how the above mentioned problems can be interpreted as pursuit-evasion problems.

- In a surveillance problem such as the Art Gallery Problem[1] [64], the guards can be represented as mobile robots equipped with a camera and tracking software.
- An evasion/detection problem where one agent wants to remain hidden/undetected from another adversarial agent can also be adapted to incorporate a robot. A robot similar to the one used in the hypothetical surveillance scenario can act as the adversary in this instance.
- Another scenario that benefits from deployable robots are search and rescue operations during a disaster. Rather than exacerbate the situation by placing the rescuers in harm's way, an alternative strategy exists where a team of autonomous search and rescue robots conduct the rescue operation. By framing the scenario in the context of a *pursuit-evasion* game where the evaders are the survivors and the pursuers are the robots, we can utilize the algorithms developed to solve pursuit-evasion problems to aid in rescue operations.

This is but a small sample of potential scenarios that illustrate the presence of pursuit-evasion problems in the real world. It is imperative that we find effective ways in which to tackle these problems.

The remainder of this introductory chapter is organized into two parts. The first is a brief overview of motion planning (Section 1.1), and the second contains a preview of the primary results and overall structure of the thesis (Section 1.2).

[1] The art gallery problem is a computational geometry problem that considers the minimum number of guards who together have the ability to observe the entire "gallery".

1.1 Motion Planning

This section is not designed to be a comprehensive guide to motion planning. Indeed, entire books [17, 46, 47] have been written on the subject. Instead, this section aims to provide enough details so that the reader is left with a clear understanding of what entails a motion planning problem, and the general tools/mechanisms used to tackle these kinds of problems.

1.1.1 Basic Ingredients of Planning

This section contains several basic ingredients that appear in motion planning. Although this thesis focuses on the pursuit-evasion problem, the following apply to nearly all motion planning problems regardless of topic.

States

A crucial idea that is the foundation of any motion planning problem is the concept of a state.

Definition 1. A *state* is the collection of all aspects of the robot and the environment that can have an impact on the future.

The state is a complete description of the robot's physical situation in the environment. A single state represents just one possible representation of the robot. The set of all possibles states is called the robot's *state space*. The notation $x \in X$ is often used to denote a specific state in the robot's state space.

Actions and Transitions

Robots interact with and move through the environment by changing their state, such changes are brought about by executing *actions*.

Definition 2. An *action*, also known as inputs or controls in control theory, is any physical interaction that causes a change to the robot's state.

An action in this case is a robot initiated physical interaction. The set of all possible actions that the robot can take is called the robot's *action space*. A robot action is typically denoted as $u \in U$, where u is a single action belonging to the robot's action space, U.

Equipped with a formal way to represent a robot's situation in the environment and a set of possible ways in which the robot can act upon the environment we can formally discuss how a robot goes about changing its state through the use of a *state transition function*.

Definition 3. A *state transition function* is a function whose input is a state x_k and an action u_k and outputs a state x_{k+1} for any time $k \geq 0$.

In its simplest form, the state transition function is a mathematical representation detailing how a robot updates its state during execution by "transitioning" from one state to another until a goal state is reached. Mathematically, the state transition function appears as

$$\begin{gathered} f : X \times U \to X \\ x_{k+1} = f(x_k, u_k). \end{gathered} \tag{1.1}$$

Observations

A central idea in Robotics is the ability for a robot to infer knowledge about itself and/or its surroundings through the use of sensors. The key idea is that any information that we want to utilize to influence the control of the robot must come from sensors. In a perfect scenario, a sensor provides complete information. This involves avoiding ambiguity by yielding complete information, utilizing noiseless sensors, and using simple enough sensors that they are easy to model. Often times sensors provide incomplete information, are noisy, and are difficult to model completely.

For this thesis, we will focus on how to represent a robot's available sensory data in the context of the models discussed in this section.

Definition 4. An *observation* is all of the current available sensor information that the robot has access to.

At its core, the observation provides the robot with a "hint" about what the current state is. An observation is typically denoted as $y \in Y$, where y is a single observation from the observation space. Since the observation is dependent on the robot's current state, the observation function appears as

$$\begin{aligned} &h : X \to Y \\ &y = h(x). \end{aligned} \tag{1.2}$$

Note that the robot need not necessarily know the state used to generate a particular observation.

Representing the Passage of Time

There are a number of ways to represent the passage of time, the following three representations are the most prevalent: continuous time, discrete time with a fixed length, and discrete stage with variable length. The choice of model is often dependent on the application. We focus on the continuous time and discrete stage models in this thesis.

In the continuous time model, time is represented by a real number t. The robot can potentially change its action at any instant in time, thus the robot's actions can be expressed as a function $u(t)$ of time. The state transition function in the continuous time domain takes the derivative of the state with respect to time and appears as $\frac{dx}{dt} = f(x, u)$ or alternatively $\dot{x} = f(x, u)$. The new state can then be computed via integration $x(t) = \int\limits_{0}^{t} f\Big(x(s), u(s)\Big)ds$.

In the discrete stage time model, time progresses in a series of discrete stages, that need not be of equal duration. This model differs from the continuous model in that time is represented by a stage counter k rather than a physical time. This higher level model allows for more abstract actions in the sense that you can perform an action u for any desired duration. The transition function for the discrete stage model is essentially the same one introduced in Equation 1.1, that is $x_{k+1} = f(x_k, u_k)$.

Representing Uncertainty

In this context *uncertainty* refers to the situation where the robot is unsure of its current state. Uncertainty in a system can be attributed to several factors; uncertain actions, uncertain sensing, and initial uncertainty. We model this ambiguity by introducing an adversary called “nature” which interferes with the robot by adding noise to the system. In this way we can adapt the models used up to this point by introducing nature actions and observations to account for any uncertainty that may exist in our system. Nature actions are denoted as $\theta \in \Theta$, where Θ is the nature action space. The introduction of nature actions requires the following modification to our state transition function in Equation 1.1:

$$\begin{aligned} f : X \times U \times \Theta \to X \\ x_{k+1} = f(x_k, u_k, \theta_k). \end{aligned} \tag{1.3}$$

Two reasonable models for interpreting the value of θ that nature chooses are nondeterministic and probabilistic models. The nondeterministic model can be viewed as the worst-case scenario model where we don’t know anything about how θ is chosen, whereas the probabilistic model assumes that nature chooses θ according to some probability distribution.

We can also assume that nature interferes with our robot’s observations as well. Nature observations are denoted as $\psi \in \Psi$, where Ψ is the nature observation space. Similar to nature actions, nature observations require modifying the observation function in Equation 1.2 to:

$$\begin{aligned} &h : X \times \Psi \to Y \\ &y = h(x, \psi) \end{aligned} \tag{1.4}$$

Our time models can be adapted to account for the effects of nature. The action errors cause the continuous time state transition function to become $\dot{x} = f(x, u, \theta)$ and the discrete stage state transition function to become $x_{k+1} = f(x_k, u_k, \theta_k)$. Similarly, sensing errors require a modification to the observation functions. The continuous time observation function becomes $y(t) = h\big(x(t), \psi(t)\big)$ and the discrete stage observation function becomes $y_k = h(x_k, \psi_k)$.

Configuration Spaces

This section provides a high level introduction to *configuration spaces*. The key idea is that we want to represent the robot as a single point in the appropriate space, and by extension seek a mapping that transforms obstacles into an appropriate representation in this space. This space is called the robot's configuration space, also commonly referred to as the C-space. A *configuration* is a single point in the configuration space.

Definition 5. A *configuration* is a complete specification of the position of every point in the system.

By considering a problem in the robot's C-space, we can transform the problem of planning the motion of a spatial object into the problem of planning the motion of a point. At its core, the C-space is just a special kind of state space.

Now we attempt to provide some intuition concerning the transformation of a general path planning problem into a problem in the robot's C-space. Informally, we want to shrink the robot down to a point and expand the obstacles by the same amount. In this way we can ensure that if the point representing our robot stays out

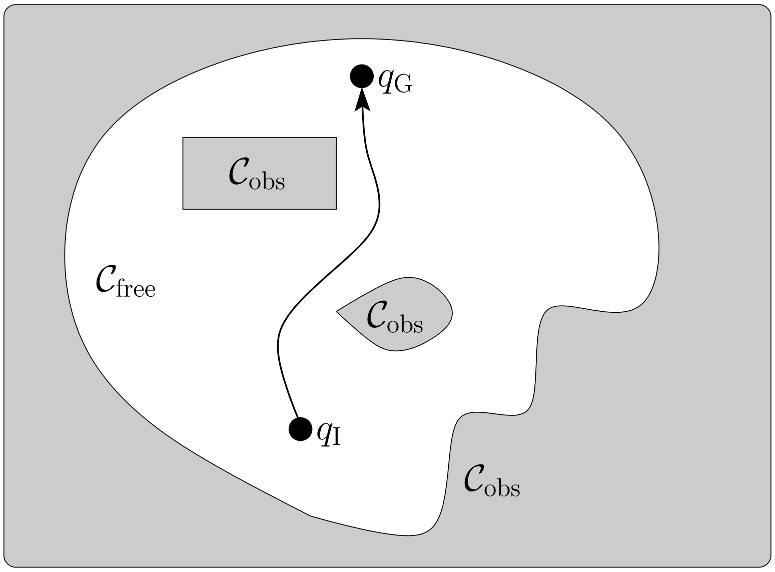

q_{G}
$\mathcal{C}_{\mathrm{obs}}$
$\mathcal{C}_{\mathrm{free}}$
$\mathcal{C}_{\mathrm{obs}}$
q_{I}
$\mathcal{C}_{\mathrm{obs}}$

Figure 1.1: The basic motion planning problem visualized using the concept of configuration space. The task is to find a collision free path in $\mathcal{C}_{\mathrm{free}}$ from q_{I} to q_{G}.

of the expanded obstacle region in the C-space, then the corresponding spatial robot will stay out of the real workspace obstacles.

The C-space is just a collection of possible configurations that our robot could occupy. However, similar to the physical domain there are areas of the C-space called *obstacle configurations* that we want to restrict the robot from entering. An obstacle configuration is a configuration in which the robot is in collision with an object in the environment. The set of obstacle configurations is denoted by $\mathcal{C}_{\mathrm{obst}}$. Everything else in the C-space is considered a *free configuration* and is denoted by $\mathcal{C}_{\mathrm{free}} = \mathcal{C} - \mathcal{C}_{\mathrm{obst}}$. Informally $\mathcal{C}_{\mathrm{free}}$ are areas corresponding to "safe" configurations for the robot.

Using the C-space formalization, the basic motion planning problem takes as input a starting configuration q_{I}, a goal configuration/region $q_{\mathrm{G}} \in \mathcal{C}_{\mathrm{goal}} \subseteq \mathcal{C}_{\mathrm{free}}$ and outputs a collision free path through $\mathcal{C}_{\mathrm{free}}$ from the starting configuration to the goal configuration/region. An illustrative example of this concept appears in Figure 1.1.

For a more mathematically rigorous definition of the configuration space we refer the reader to Chapter 4 of LaValle's text on Planning Algorithms [47].

Information Spaces

This section provides the reader with a high level understanding of *information spaces*. The key idea, is that in the presence of state uncertainty where the robot's true state is hidden, we want to create some representation based on the information available to the robot. Fundamentally, information spaces can be viewed as a special kind of state space.

What information is available to the robot? In some scenarios, initial conditions may be specified in such a way that the initial state may be known, but in general a robot has access only to the history of past actions and the sensor observations it has received. The space of past histories is called the robot's *history information space* and is denoted by $\mathcal{I}_{hist}$ and defined as

$$\mathcal{I}_{hist} = \bigcup_{i=0}^{\infty} (U \times Y)^i. \tag{1.5}$$

After k stages, the robot's history information state is the following sequence of action-observation pairs

$$\eta_{k+1} = (u_1, y_1, \ldots, u_k, y_k) \in \mathcal{I}_{hist}. \tag{1.6}$$

The history I-space provides a way of storing and maintaining the action-observation histories, but does not provide any insight into how the robot might make use of this information. It typically is not feasible to deal with history I-states explicitly because the length of a history I-state grows linearly with the number of stages. Instead, we consider information mappings (I-maps) of the form

$$\kappa : \mathcal{I}_{hist} \rightarrow \mathcal{I} \tag{1.7}$$

that consolidate the history I-states into a new target space $\mathcal{I}$ called a *derived information space*. Informally, κ can be viewed as the mechanism that the robot uses to interpret its sensor information. Naturally the usefulness of a derived I-space is dependent on the mapping's ability to capture relevant information.

For the purpose of reasoning about planning problems, we consider a special class of I-maps called *sufficient I-maps*. Given an I-map $\kappa : \mathcal{I}_{hist} \to \mathcal{I}$, κ is a sufficient I-map if there exists an information transition function

$$f_{\mathcal{I}} : \mathcal{I} \times U \times Y \to \mathcal{I} \tag{1.8}$$

such that

$$f_{\mathcal{I}}\Big(\kappa(\eta_k), u_k, y_k\Big) = \kappa(\eta_k, u_k, y_k) \tag{1.9}$$

for any $\eta_k \in \mathcal{I}_{hist}, u_k \in U, y_k \in Y$. The intuition is that the I-states derived by κ are sufficient to determine future derived I-states. This is very similar to the idea of a sufficient statistic [13, 90] in Statistics. In this sense the current derived I-state is as powerful as having the complete action-observation history when computing future derived I-states. So we are able to reason about the problem in the derived I-space rather than the history I-space.

It remains to show what a solution to a planning problem looks like in an information space. First, consider how the goal region in an information space differs from that of a typical state space. A goal region in a state space is the set of terminating configurations that the robot could be in when it satisfies its task, whereas a goal region in an information space must account for all of the potential action-observation pairs that could cause the robot to enter into a configuration that accomplishes the task. So naturally we can represent the goal region, $\mathcal{I}_G$ as a subset of the history I-space.

The last piece of information that we need is a mechanism that guides our search through the I-space to the goal region. In essence we want a mapping

$$\pi : \mathcal{I} \to U \tag{1.10}$$

that, given an I-state, selects the next action the robot will take. Such a mapping is called a *policy* over a derived I-space, and if repeated applications of π produces an I-state in $\mathcal{I}_G$ then π is a *solution* to the problem.

1.1.2 Properties of Planners

This section introduces several different properties of planners focusing on the quality of the solution they provide. In particular we focus on *optimality* and varying degrees of *completeness*. We use the following definition to define what it means for a planner to be *optimal*.

Definition 6. A planner is *optimal* if it finds motions that optimize some parameter such as length, execution time, or energy consumption.

This definition places the onus on the author to clearly specify the parameter(s) being optimized. This often aids the reader by providing some insight about the problem and/or the robot model. The following example demonstrates this idea.

Example: A planner that generates a minimal cost (Euclidean distance) path.

False Assumption: The minimal cost path will **also be** the path that minimizes the robot's execution time to follow the aforementioned path.

Scenario: This case often arises when the path requires the robot to spend a large amount of time rotating as opposed to translating.

We use the following definition to define what it means for a planner to be *complete*.

Definition 7. A planner is *complete* if it will always find a solution to the motion planning problem when one exists, or indicate failure in finite time if no solution exists.

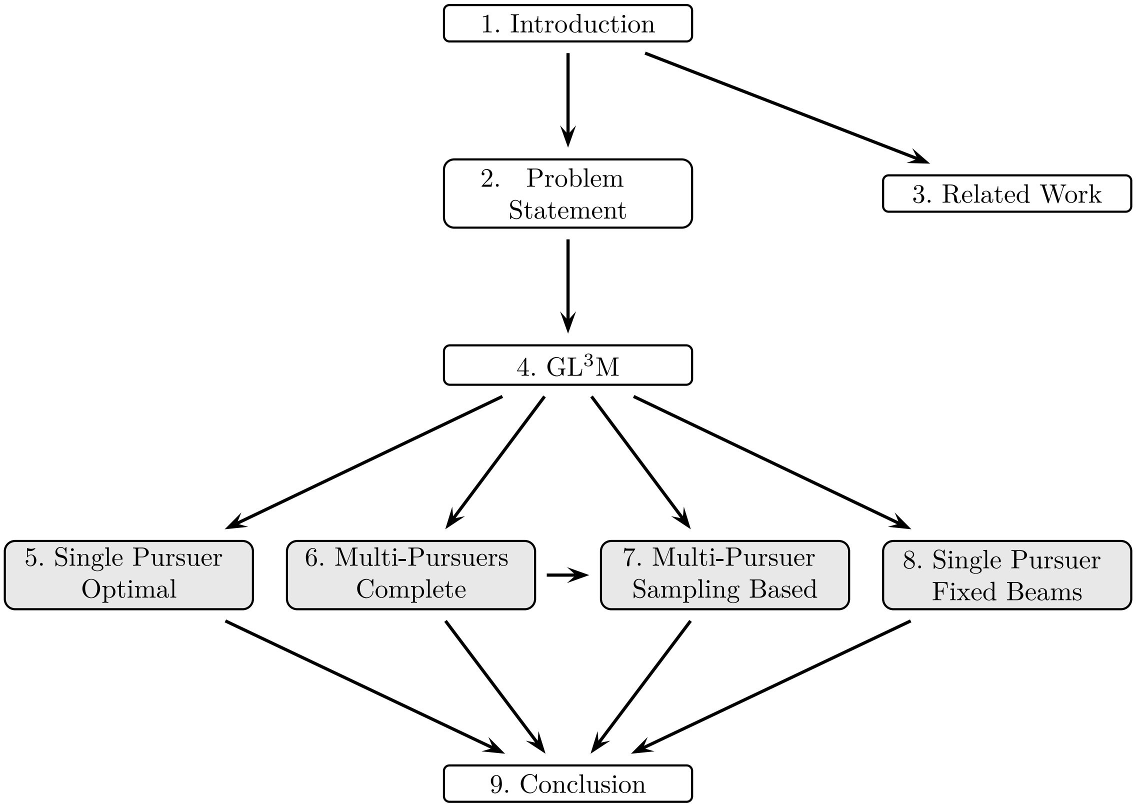

1. Introduction
2. Problem Statement
3. Related Work
4. GL^{3}M
5. Single Pursuer Optimal
6. Multi-Pursuers Complete
7. Multi-Pursuer Sampling Based
8. Single Pursuer Fixed Beams
9. Conclusion

Figure 1.2: Organization of this thesis with arrows indicating dependencies. Novel results are denoted by the shaded blocks.

While complete algorithms are desirable, they become intractable as the degree of the configuration space gets larger [11]. Therefore we often seek weaker forms of completeness such as *resolution* completeness or *probabilistic* completeness.

Definition 8. A planner is *resolution complete* if a solution exists at a given level of discretization. A planner is *probabilistically complete* if the probability of finding a solution tends to 1 as time goes to infinity.

1.2 Thesis Organization

We conclude this introductory chapter with a preview of the remainder of this thesis. A formal problem statement appears in Chapter 2. A literature review appears in Chapter 3. Chapter 4 contains an overview of the Guibas, Latombe, LaValle, Lin, and

Motwani (GL^{3}M) algorithm that influenced the four novel contributions that appear in Chapters 5, 6, 7, and 8. Concluding remarks and some potential avenues for future work appear in Chapter 9. The structure and dependencies between chapters are shown in Figure 1.2. This thesis presents four novel results for various visibility-based pursuit-evasion problems: an optimal search strategy for a single pursuer, a complete algorithm for multiple pursuers, a randomized algorithm for multiple pursuers, and a complete algorithm for a single pursuer with limited sensing capabilities.

1.2.1 Single Pursuer - Shortest Path

The first result (Chapter 5) considers an instance of the visibility-based pursuit-evasion problem that utilizes a single pursuer to search the environment for potential evaders. The main contribution is a complete algorithm whose goal is to compute a minimal-cost pursuer trajectory that ensures that the evaders are captured in a finite time, or reports that no finite time pursuer trajectory exists. This result improves upon the known algorithm of Guibas, Latombe, LaValle, Lin, and Motwani, which is complete but makes no claims as to the quality of the solution. The central idea is that by carefully decomposing the two-dimensional polygonal environment into combinatorially equivalent convex regions, we can exploit the structure of the problem by considering a simpler subproblem that is equivalent to computing the minimal-cost pursuer trajectory.

1.2.2 Multiple Pursuers - Complete Solution

The second result (Chapter 6) considers an instance of the visibility-based pursuit-evasion problem that utilizes multiple pursuers to search an environment. We present a centralized algorithm that searches the pursuers' joint configuration space for a joint strategy for the pursuers that will satisfy the capture conditions of the pursuit-evasion problem. The main idea is to construct a Cylindrical Algebraic Decomposition(CAD) of the pursuers' joint configuration space by using polynomials that capture where critical changes can occur to the region of the environment hidden from the pursuers. After computing the adjacency graph for the CAD, we construct a Pursuit-Evasion Graph(PEG) induced by the adjacency graph. A search through the PEG can produce one of the following outcomes; the search can reach a vertex where the pursuers' motions up to this point ensures that the evader has been captured, or the search terminates without finding a solution and produces a statement recognizing that no solution exists.

1.2.3 Multiple Pursuers - Probabilistically Complete Sampling-Based Solution

Motivated by the complexity of the previous result, our third result (Chapter 7) introduces a probabilistically-complete sampling-based algorithm for solving a visibility-based pursuit-evasion problem that utilizes multiple pursuers. This technique constructs a Sample-Generated Pursuit-Evasion Graph (SG-PEG) that utilizes an abstract sample generator to search the pursuers' joint configuration space for a search strategy that captures the evaders, or reports that no such strategy exists under the current constraints.

1.2.4 Single Pursuer - Fixed Beams

The final result (Chapter 8) considers an instance of the visibility-based pursuit-evasion problem where a single pursuer is equipped with a finite collection of single-direction sensors, with the goal of locating an adversarial evader within the line-of-sight of one of those sensors. The novel contribution is a complete and efficient algorithm for solving this fixed-beam pursuit-evasion problem. The intuition of the algorithm is to decompose the environment into a collection of convex conservative regions, within which the evader cannot “sneak” between any pair of adjacent sensors. This decomposition induces a graph we call the Fixed-Beam Pursuit-Evasion Graph (FB-PEG), such that any correct solution strategy can be expressed as a path through the FB-PEG.

Chapter 2

Problem Statement

This chapter formalizes the visibility-based pursuit-evasion problem considered in this thesis. We begin by describing the model used to represent the environment, evaders, and pursuers (Section 2.1) and then give a formal definition for the area of the environment not visible to the pursuers, called shadows (Section 2.2).

2.1 Representing the environment, evaders, and pursuers

The environment is a polygonal free-space, defined as a closed and bounded set $W \subseteq \mathbb{R}^2$, with a polygonal boundary ∂W. The boundary of the environment is composed of m vertices.

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 is capable of moving arbitrarily fast (i.e. a finite, unbounded speed) within W. The evader trajectory e is unknown to the pursuers. Without loss of generality we can assume that there is a single evader. If the pursuers can guarantee the capture of a single evader, then the same strategy can locate multiple evaders, or confirm that no evaders exist.

A collection of n identical pursuers cooperatively move to locate the evader. We assume that the pursuers know W, and that they are centrally coordinated. Therefore, from a given collection of starting positions, the pursuers' motions can be described by a continuous function $p : [0, \infty) \rightarrow W^n$, so that $p(t) \in W^n$ denotes the joint configuration of the pursuers at time $t \geq 0$. The function p is called a *joint motion strategy* for the pursuers. We use the notation $p^i(t) \in W$ to refer to the position of pursuer i at time t. Likewise, $x^i(t)$ and $y^i(t)$ denote the horizontal and vertical coordinates of $p^i(t)$. Without loss of generality, we assume that the pursuers move with maximum speed 1.

Each pursuer carries a sensor that can detect the evader. The sensor is omnidirectional and has unlimited range, but cannot see through obstacles. For any point $q \in W$, 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.

When considering the maximal path-connected component of $V(q)$, the edges of its boundary are either along ∂W or belong to an occlusion ray.

Definition 9. An *occlusion ray*, $\overrightarrow{qr}$, is a ray starting at a pursuer position q 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 group of pursuers executing the joint motion strategy p is denoted as:

$$t_c(p, e) = \min \left\{ t \geq 0 \;\middle|\; e(t) \in \bigcup_i V\left(p^i(t)\right) \right\} \tag{2.1}$$

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

Definition 10. A pursuer joint motion strategy p is a *solution strategy* if there exists a finite time of capture, denoted $t_c(e)$ and defined as

$$t_c(p) = \max_e \; t_c(p, e). \tag{2.2}$$

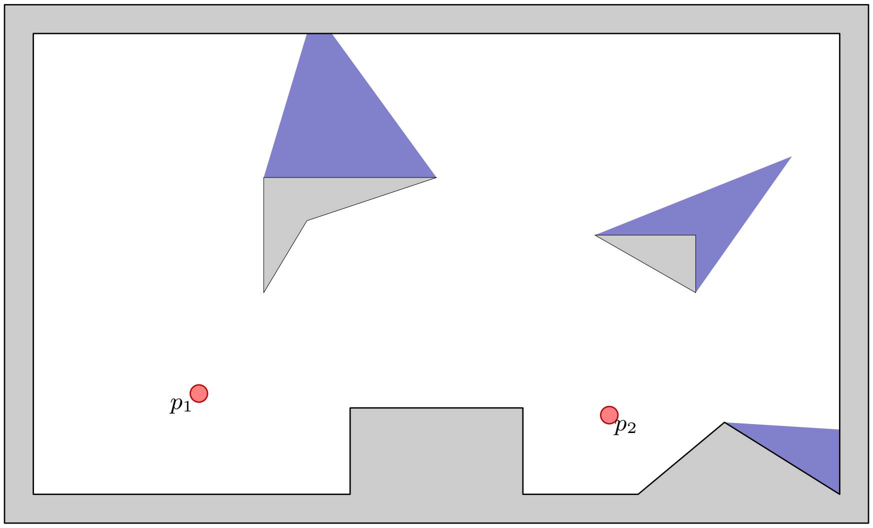


Figure 2.1: An environment with two pursuers (red circles) and three shadows (filled path-connected regions).

The time $t_c(e)$ is the least upper bound for the time of capture over all valid evader trajectories when the pursuers follow the joint motion strategy p.

2.2 Shadows

The key difficulty in locating the evader is that the pursuers 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 pursuers at any particular time.

Definition 11. The portion of the environment not visible to the pursuers at time t is called the *shadow region* $\mathcal{S}(t)$, and defined as

$$\mathcal{S}(t) = W - \bigcup_{i=1,\ldots,n} V\big(p^i(t)\big).$$

Note that the shadow region may contain zero or more nonempty path-connected components, as seen in Figure 2.1.

Definition 12. 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.

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

- *Appear*: A new shadow can appear, when a previously visible part of the environment becomes hidden.
- *Disappear*: An existing shadow can disappear, when one or more pursuers move to locations from which that region is visible.
- *Split*: A shadow can split into multiple shadows, when the pursuers move in such a way that a given shadow is no longer path-connected.
- *Merge*: Multiple existing shadows can merge into a single shadow, when previously disconnected shadows become path-connected.
- *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 were originally enumerated in the context of the single-pursuer version of this problem [27] and examined more generally by Yu and LaValle [96].

2.2.1 Shadow Labels

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

Definition 13. A shadow s is called *cleared* at time t if, based on the pursuers' motions up to time t, it is not possible for the evader to be within s without having been captured.

Definition 14. 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 pursuers cannot draw any more detailed conclusion about each shadow than its clear/contaminated status; if any part of a shadow might contain the evader, then the entire shadow is contaminated. Using this worst-case reasoning, we can completely represent the I-state given the pursuers' current configuration and the current shadow labels.

Comparison Operators: Equal (=) and Not Equal ($\neq$)

This section describes two comparison operators for shadow labels that test for equality. Consider two shadow labels $S = (s_1 s_2 \ldots s_k)$ and $S' = \left(s_1^{'} s_2^{'} \ldots s_k^{'}\right)$.

Definition 15. Two shadow labels S and S' are *equal to*(=) one another if the following holds:

$$\forall i \quad 1 \leq i \leq k \quad : \quad s_i = s_i^{'}.$$

The intuition behind the (=) relation is that if a shadow appears as cleared in S then it must also be cleared in S'. Similarly, if a shadow is contaminated in S then it must also be contaminated in S'.

Definition 16. Two shadow labels S and S' are *not equal to* one another if the following holds:

$$\exists i \quad 1 \leq i \leq k \quad : \quad s_i \neq s_i^{'}.$$

The intuition behind the ($\neq$) relation is that there must exist at least one shadow whose label is dissimilar between S and S'. The not equal relation is the logical negation of the equal to relation.

Binary Relation: Dominates ($\gg$)

This sections describes a dominance binary relation over shadow labels. Consider two shadow labels $S = (s_1 s_2 \ldots s_k)$ and $S' = \left(s_1^{'} s_2^{'} \ldots s_k^{'}\right)$.

Definition 17. A shadow label S *dominates* a shadow label S' if the following holds:

$$\forall i \quad 1 \leq i \leq k \quad : \quad s_i \leq s_i^{'}.$$

Informally, S dominates S' if for every shadow that is cleared in S', the corresponding shadow in S is also cleared. The intuition is that S provides at least as much information as S', and can potentially contain more information in the case where $s_i = 0$ and $s_i^{'} = 1$.

Definition 18. A shadow label S *strictly dominates* ($\underline{\gg}$) a shadow label S' if

$$S \gg S' \quad \textbf{and} \quad S \neq S'. \tag{2.3}$$

2.2.2 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 visibility event that has occurred. Each rule describes how a label preceding the visibility event is updated immediately following a given visibility event.

- *Appear*: New shadows are formed from regions that had just been visible, so they are assigned a clear label.

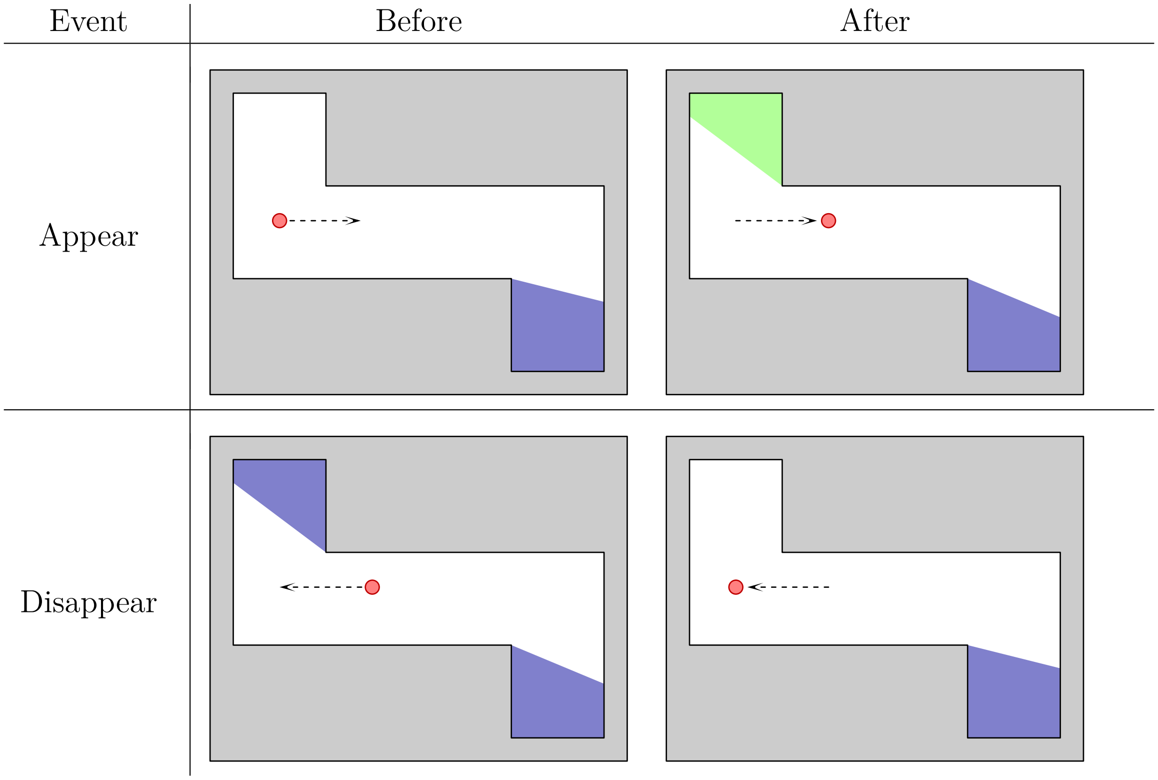


Figure 2.2: An appear event increases the number of shadows by one, and the new shadow is labelled clear (green region). A disappear event decreases the number of shadows, its label is discarded.

- *Disappear*: When a shadow disappears, its label is discarded.
- *Split*: When a shadow splits, the new shadows inherit the same label as the original.
- *Merge*: When shadows merge, the new shadow is assigned the worst label of any of the original shadows' labels. That is, a shadow formed by a merge event is labeled clear if and only if all of the original shadows were also clear.
- *Push*: When a shadow is pushed, it maintains its current label.

Figures 2.2, 2.3, and 2.4 illustrate the shadow label update rules where cleared shadows are represented as the filled path-connected green regions and contaminated shadows are represented as the filled path-connected purple regions.

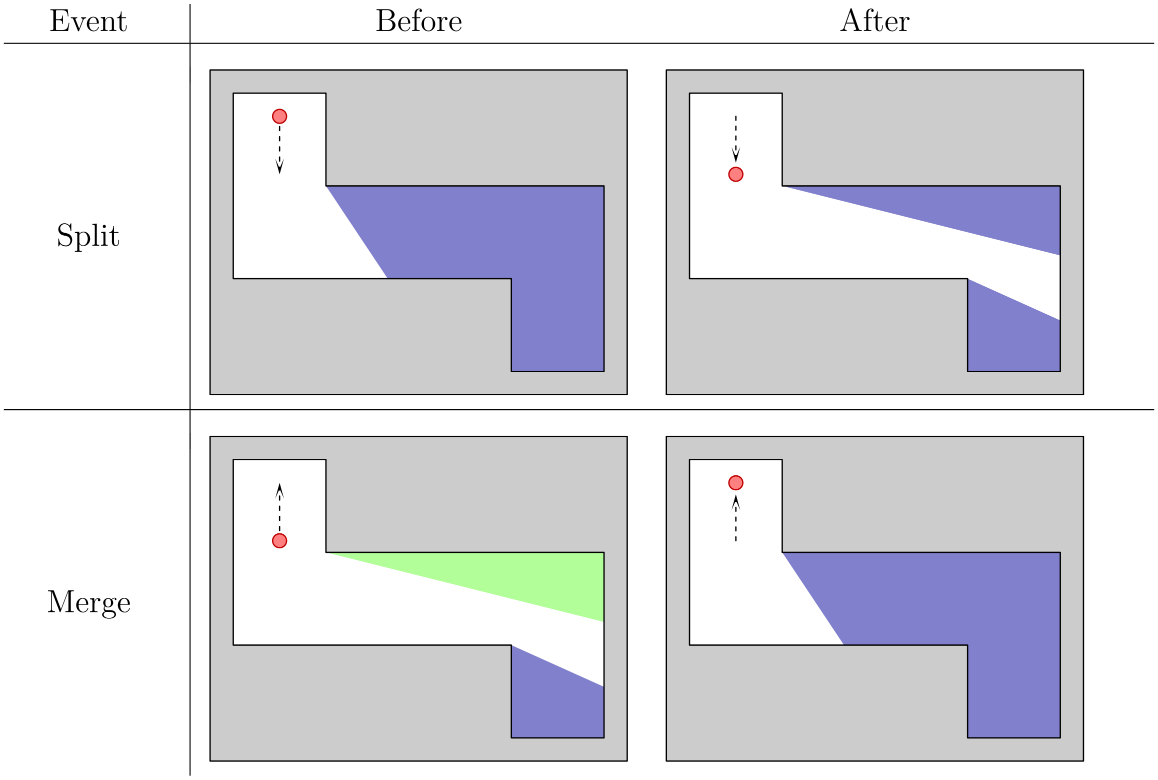


Figure 2.3: When a shadow splits into multiple shadows, they inherit the same label as the original shadow. When a merge event occurs the new shadow is clear if and only if all of the original shadows are also clear.

2.3 Reformulating the 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 10 where the pursuers' goal was stated as computing a finite time of capture for each 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 in the event that all of 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 19. A pursuer joint motion strategy is a *solution strategy* if and only if it

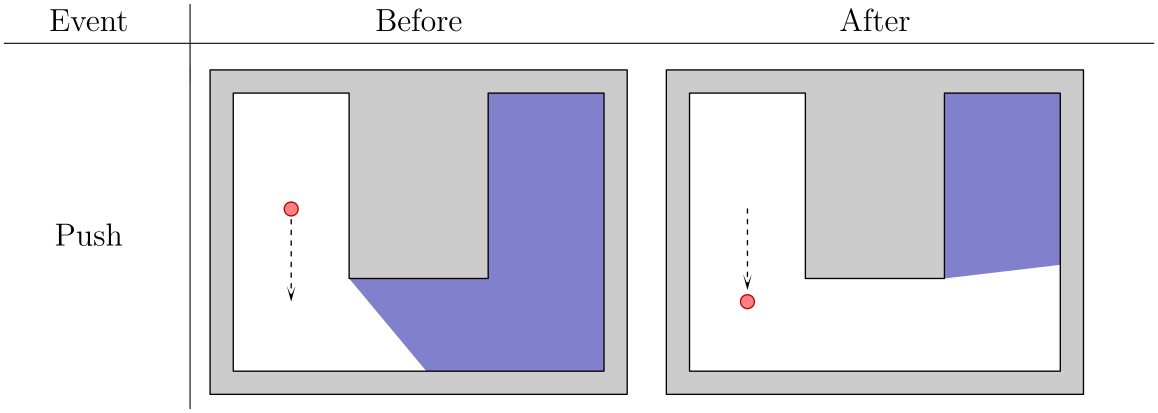


Figure 2.4: A push event occurs when a shadow gets pushed between neighboring pairs of environment reflex vertices.

reaches a pursuer configuration in finite time in which all of the shadows are cleared.

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

Chapter 3

Related Work

Pursuit-evasion problems are often classified under the vast umbrella of target tracking, security, and monitoring and surveillance problems. For brevity, this thesis will use the term *target tracking* to refer to this collection of similar problems. Target tracking problems typically require the system to ascertain some information about a target (environmental feature and/or a mobile agent). Target tracking problems span multiple disciplines and can be found in game theory [70,78], computational geometry [12,15,16,53], wireless networks [14,25,28], and mobile robotics [4,33]. In this chapter we provide a brief overview of the target tracking problem (Section 3.1) followed by a more focused literature review of Pursuit-Evasion problems (Section 3.2).

3.1 Target Tracking

As mentioned above, target tracking is a problem that spans multiple disciplines. This thesis focuses on those works most closely related to mobile robotics. In the most general case the objective for these problems is to maintain visibility between the target and the tracker. Algorithms are known for planning the tracker's motions using dynamic programming [48], sampling-based methods [57], Partially Observable Markov Decision Process (POMDP) [30], and reactive approaches [55]. The target tracking task can be further complicated by additional constraints such as avoiding detection [4], maintaining the target's privacy [60], and bounded observer speed [54].

The remainder of this section focuses on two related target tracking problems.

The first problem is wireless sensor network assisted target tracking (Section 3.1.1) and the second problem is monitoring and surveillance (Section 3.1.2). This is a small sample of the various tasks and approaches encompassed within target tracking.

3.1.1 Wireless Sensor Network Target Tracking

As mentioned above, wireless sensor networks (WSNs), have been one approach used to track a moving target(s). This target could be a human [14, 80], a moving vehicle [26, 28, 97], or other moving target [3, 40, 79]. WSNs have been used in conjunction with mobile robots [33, 62] to tackle the target tracking problem. Typically, the mobile robots are used during sensor deployment with the goal of achieving good sensor coverage [5,6]. However, their has been work done that focuses on the tracking application after the deployment has occurred [63].

3.1.2 Monitoring and Surveillance

Monitoring and surveillance are two terms that are often used interchangeably. The key distinction is that monitoring is a passive task that does not result in any direct action on the agent's part. The monitoring task typically charges the agent with using its sensory information to detect a change in its environment. Surveillance tasks are often seen as the active version of the monitoring problem where an agent is tasked with actively searching its environment in an effort to detect some change in the environment.

Persistent monitoring and surveillance tasks are variations on the traditional monitoring and surveillance tasks that require a tracker or team of trackers to perform their monitoring/surveillance task in perpetuity. The "perpetuity" aspect is what makes these problems well-suited to be carried out by a robot or robot team. Visibility-based monitoring problems commonly occur in many applications such as security and surveillance [91], infrastructure inspection [65], and environmental monitoring [81].

The increased availability of mobile robots capable of performing these tasks has led to increased interest [45, 51, 82] in recent years.

3.2 Pursuit-Evasion

This section examines existing literature in the field of pursuit-evasion. Although this thesis presents results for a visibility-based pursuit-evasion problem, we discuss the evolution of the pursuit-evasion problem from differential games (Section 3.2.1) to a graph-based formulation (Section 3.2.2) and finally to a geometric formulation (Section 3.2.3).

3.2.1 Differential Games

The pursuit-evasion problem was originally posed in the context of differential games [29, 31] and has produced a variety of different problems with small variations. In the lion and man game, a lion tries to capture a man who is trying to escape [37, 58, 59, 77, 93]. 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 [31, 74]. Bounds for this problem that require the pursuer to physically capture the evader suggests the number of pursuers required to satisfy this capture condition exceeds that needed for the visibility-based pursuit-evasion problem [41].

3.2.2 Graph-Based Formulation

Pursuit-evasion on a graph can be traced back to the independent work done by Parsons and Petrov. The motivation behind the Parsons' problem 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. The idea behind the Parsons' problem [67], also known as the edge-searching problem, is to determine a sequence of moves for the pursuers that can detect all intruders in a graph using the least number of 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.

The Parsons' problem and some of its results were later independently rediscovered by Petrov [68] using slightly different motivating problems. Petrov's formulation considered the cossacks and the robber game [69] and the princess and the monster problem [31]. Golovach showed that both problems considered an equivalent discrete game on graphs [23].

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 [44]. 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 is but a small sample of the existing literature surrounding the graph-based pursuit-evasion problem. We have placed an emphasis on the inception of the problem and briefly touched on some recent results. For a more comprehensive review of recent results in graph-based pursuit-evasion we direct the reader to the following surveys [1, 9, 10, 21, 89].

3.2.3 Geometric Formulation

The visibility-based pursuit-evasion problem and the surveillance/tracking problem are various types of pursuit-evasion problems that use a geometric formulation.

The first visibility-based pursuit-evasion problem was proposed by Suzuki and

Yamashita [88] as an extension of the watchman route problem[1] [16] and is a geometric formulation of the traditional graph-based pursuit-evasion problem. Research on the visibility-based pursuit-evasion has produced numerous results for both the single pursuer and multiple pursuer variants of the problem.

Single Pursuer Visibility-Based Pursuit-Evasion

There are many interesting results for the single pursuer visibility-based pursuit-evasion problem. A complete solution [27], a randomized solution [32], and an optimal shortest path solution have been found.

The capture condition for the general visibility-based pursuit-evasion problem is defined as having an evader lie within the pursuer's capture region. There has been substantial research focused how the visibility-based pursuit-evasion problem changes when a robot has different capture regions. The k-searcher is a pursuer with k visibility beams [50,88], the ∞-searcher is a pursuer with omni-directional field of view [27,66], and the ϕ-searcher is a pursuer whose field-of-view [22] is limited to an angle $\phi \in (0, 2\pi]$. Note that all of these approaches consider evaders with unbounded speed.

Others have studied scenarios where there are additional constraints, such as the case of curved environments [49], an unknown environment [75], a maximum bounded speed for the pursuer [92], or constraints on the pursuer similar to those of a typical bug[2] algorithm [73].

[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 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.

Multiple Pursuer Visibility-Based Pursuit-Evasion

As a result of the problem complexity, there is a wide range of literature with differing techniques attempting to solve the multi-robot visibility-based pursuit-evasion problem. Some recent results involve using some of the pursuers as stationary sentinels while other pursuers continue with the search [43]. Another approach involves maintaining complete coverage of the frontier [20]. There are other variants of the pursuit-evasion problem where the pursuers are teams of unmanned aerial vehicles [42].

Chapter 4

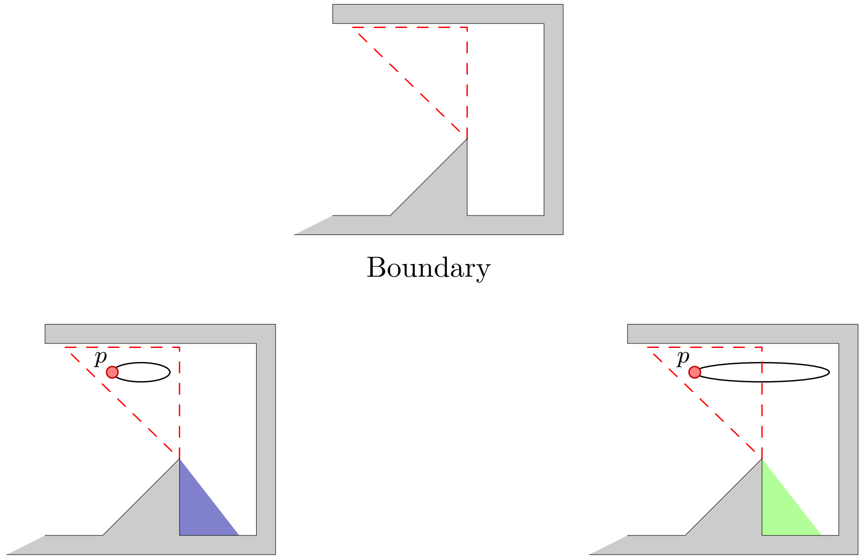
GL^{3}M Algorithm

The prior work of Guibas, Latombe, LaValle, Lin, and Motwani is integral in understanding some of the techniques that contribute to the results in this thesis. As such, it is necessary to summarize the work of Guibas, Latombe, LaValle, Lin, and Motwani [27] that presents a complete solution to the visibility-based pursuit-evasion problem that utilizes a single pursuer.

4.1 Overview

The authors' main contribution is a way to change the continuous problem of finding a pursuer trajectory into a simpler discrete problem. Initially, the problem requires a pursuer trajectory that solves the single pursuer visibility-based pursuit-evasion problem. But by considering the areas of the environment that induce changes to the shadow region we can ask the following equivalent question. What areas of the environment does the pursuer have to visit to guarantee that the evader is captured? Once a valid sequence has been found, returning a trajectory is trivial.

In the remainder of this chapter we investigate how certain pursuer motions can force a critical information change within the shadow region (Section 4.2), and describe a graph structure and algorithm for solving the single pursuer visibility-based pursuit-evasion problem (Section 4.3).



Boundary

Contaminated Shadow
Path does not cross a critical boundary

Cleared shadow
Path crosses a critical boundary

Figure 4.1: An illustration of the concept of conservative regions.

4.2 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 algorithm, the authors need only be interested in tracking times in which the pursuer's information changes *combinatorially*. That is, we are only concerned with pursuer movements that generate shadow events, as seen in Figure 4.1.

Definition 20. A region $R \subseteq W^n$ is a *conservative region* if any path that remains within R generates no shadow events.

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

The original paper describes a visibility cell decomposition of the environment that captures where the critical changes to the pursuer's I-state occur. The decomposition

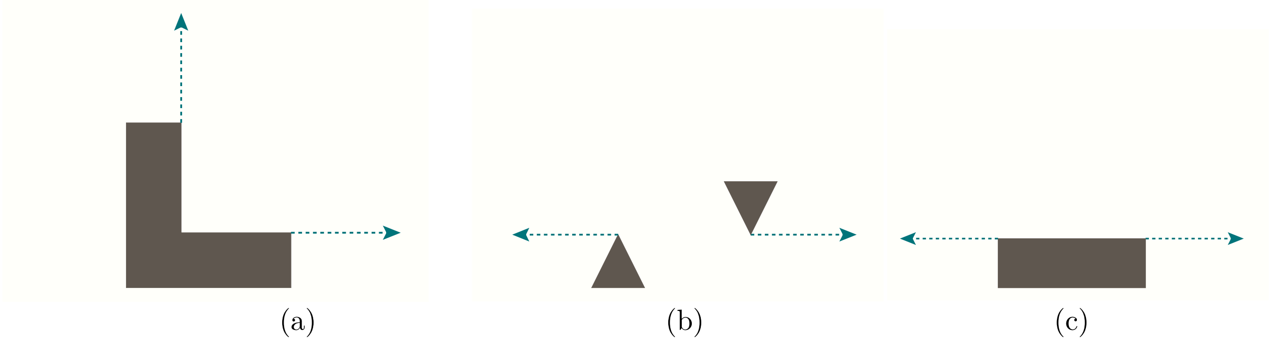

(a) (b) (c)

Figure 4.2: Ray shooting is performed for three general cases to form the conservative regions.

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 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 2.2.

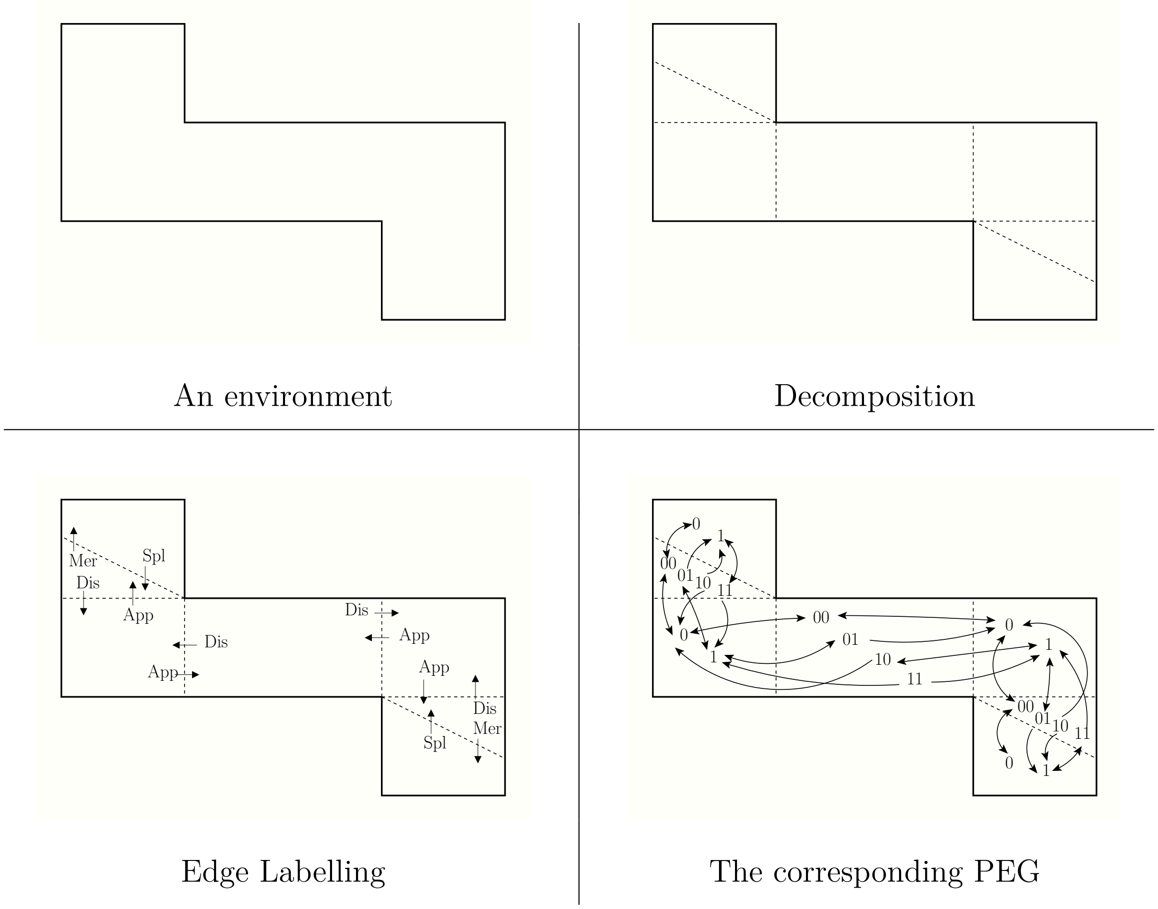
The procedure used in creating the ray extensions provides the following information about what type of event takes place along the boundary of the extension:

(a) Ray extensions caused by an inflection at a single endpoint of an environment edge cause appear and disappear events.

(b) Ray extensions caused by a pair of mutually visible environment vertices (where the vertices are not part of the same environment edge) cause split and merge events.

(c) Ray extensions caused by inflections at both endpoints of an environment edge cause push events.

Figure 4.2 illustrates the various partitioning operations.



An environment

Decomposition

Edge Labelling

The corresponding PEG

Figure 4.3: An example of the Pursuit Evasion Graph for a given environment.

4.3 The Pursuit-Evasion Graph

With this information, the complete Pursuit-Evasion Graph (PEG) can be constructed as shown in Figure 4.3. The PEG is a directed graph composed of nodes that contain a shadow labeling and a reference to a conservative region, where a node exists for each possible shadow label combination for every conservative region. Its edges are the set of critical events that occur from crossing an event boundary from one conservative region to another. The algorithm starts at the PEG-node that contains $p(0)$ with a shadow label of $1 \cdots 1$. Using this node as the root of a graph search, the algorithm uses breadth-first search to find a path to a node with a shadow label of $0 \cdots 0$. This path through the PEG provides a sequence of conservative regions to visit. The algorithm then constructs a path through W by moving to the centroid of each conservative region that appears in the sequence.

Chapter 5

An Optimal Strategy for a Single Pursuer

The specific problem we consider in this chapter is a variation on the visibility-based pursuit-evasion problem in which a single pursuer moving through a simply-connected polygonal environment seeks to locate an unknown number of evaders, each of which may move arbitrarily fast. The pursuer has an omni-directional field-of-view that extends to the environment boundary.

The goal is to compute a pursuer strategy such that all evaders in the environment lie within the pursuer's field-of-view at some finite time as the pursuer carries out its search strategy, or to identify when no such strategy exists. Guibas, Latombe, LaValle, Lin, and Motwani presented a complete algorithm for this problem [27], the details of which appear in Chapter 4. However, the authors consider only feasibility and do not attempt to compute optimal strategies. We build upon this work by developing an algorithm that solves the visibility-based pursuit-evasion problem by returning a solution strategy that is optimal in the sense that it minimizes the distance travelled by the pursuer.

We use the same decomposition and Pursuit-Evasion Graph (PEG) discussed in Section 4.2 and Section 4.3, but our algorithm must simultaneously consider multiple paths to each node. Each of these paths can be viewed as a tour that travels through an ordered sequence of cell boundaries. We introduce a pruning operation to eliminate suboptimal paths, and a forward search algorithm whose termination condition guarantees that an optimal solution will be found.

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

The remainder of this chapter is structure as follows. Section 5.1 formalizes the objective. Section 5.2 introduces an algorithm for computing the shortest path that visits a given sequence of segments in order. Section 5.3 describes an algorithm that either returns the optimal pursuer solution strategy or is able to recognize that no such strategy exists. Section 5.4 presents our simulations of this algorithm and a quantitative illustration of its effectiveness.

A preliminary version of this work appears in [84].

5.1 Formalizing the Objective

For clarity we will explicitly define the time of capture for a single pursuer. Note that Equation 5.1 is just a special case of Equation 2.1 when there is only a single pursuer.

Definition 21. The time of capture for an **individual** evader following trajectory e and a single pursuer following trajectory p is denoted as

$$t_c(p,e) = \min\Big\{t \geq 0 \quad | \quad e(t) \in V\big(p(t)\big)\Big\}. \tag{5.1}$$

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

Definition 22. A pursuer trajectory p is a *solution strategy* if there exists a finite time of capture, denoted $t_c(p)$ and defined as

$$t_c(p) = \max_e \;\; t_c(p,e). \tag{5.2}$$

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

$$p^* = \underset{p}{\operatorname{argmin}} \Big(t_c(p)\Big).$$

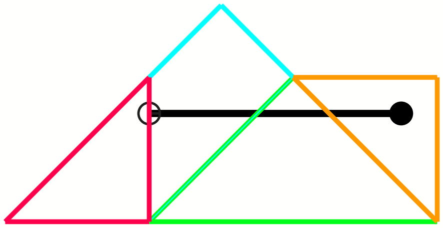
Our goal is to compute this optimal pursuer strategy p^*.

5.2 Optimal Tours of Segments

This section describes the subroutine we use to solve the subproblem of computing the shortest path that traverses a sequence of conservative regions. This problem requires us to do the following:

Given: A point p, and a sequence of conservative regions $(c_1, \ldots, c_n)$.

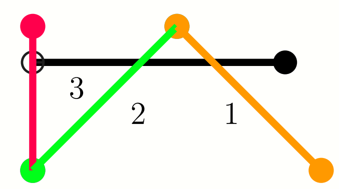
Compute: The shortest path that starts at p and visits the conservative regions $(c_1, \ldots, c_n)$ in order.



Using the decomposition described in Section 4.2 we are guaranteed to have a partitioning of the environment into convex conservative regions. So the subproblem is equivalent to solving a *Tour of Polygons* problem where the polygons are the conservative regions. The problem can be simplified even further by taking advantage of one of the properties of our decomposition, namely the fact that each conservative region boundary edge is shared with only a single corresponding conservative region. Informally, this means that there exists only a single edge belonging to any pair of neighboring conservative regions. This means that we can restate the problem as:

Given: A point p, and an ordered collection of segments $(s_1, \ldots, s_{n-1})$.

Compute: The shortest path that starts at p and visits the segments $(s_1, \ldots, s_{n-1})$ in order.



This is a simpler version of the Tour of Polygons problem known as a *Tour of Segments*.

Definition 23. Given a point p and an ordered collection of segments $(s_1, \ldots, s_n)$, the shortest path that starts at p and visits the segments $(s_1, \ldots, s_n)$ in order is called a *Tour of Segments* (TOS).

Dror, Efrat, Lubiw, and Mitchell showed how to compute such paths in a more general case in which the intermediate steps are polygons rather than segments [19]. We adapt this approach for the specific case of a sequence of segments.

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

5.2.1 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 subdivides the plane into four 2-dimensional cells, five 1-dimensional cells, and two 0-dimensional cells. Figure 5.1 shows an example. The key idea is that all shortest paths starting from p to all points in one of the aforementioned cells will have an equivalent combinatorial structure.

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

SPM: 1-dimensional cells There are five 1-d cells in a SPM denoted as A, B, C, D, and E. The 1-d cells are constructed from segment s and the start point p. One of these 1-d cells is an open line segment and corresponds to s (excluding the endpoints) whereas the four remaining 1-cells are all open rays, two originating from left(s) and two originating from right(s). The following describe each of the 1-d 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: 2-dimensional cells There are four 2-d cells in a SPM that are separated by the 1-d cells. The following describes which 1-cells form the boundary of our 2-cells.

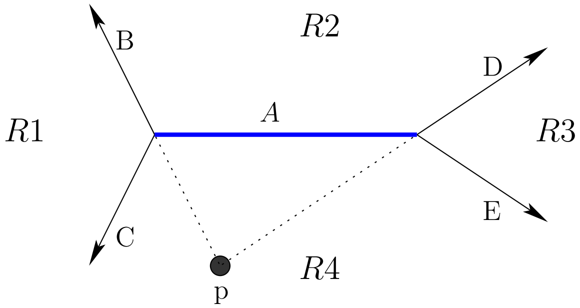


Figure 5.1: A single Shortest Path Map. These four rays and one segment subdivide the plane into regions with combinatorially equivalent shortest paths.

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

5.2.2 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 5.2. There are four general cases which correspond to the 2-d 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.

(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.

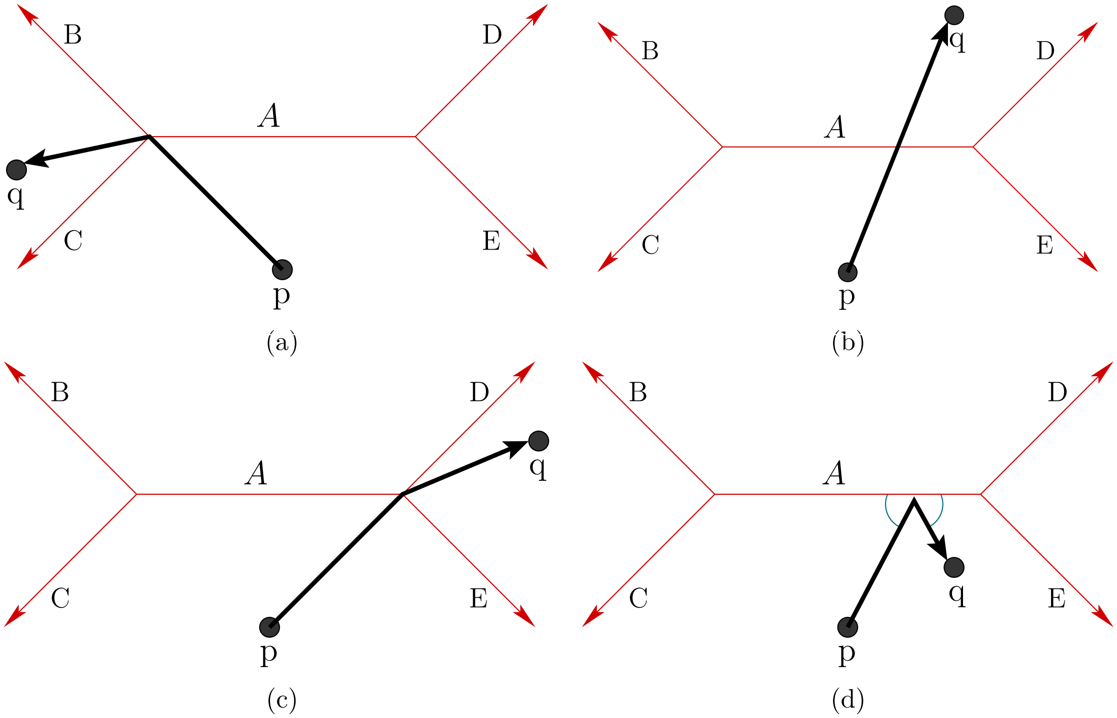


Figure 5.2: The SPM for the first segment s divides the plane according to the combinatorial structure of the shortest path from p to s to a query point q.

(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 of s.

We have described the procedure for creating an single SPM, however when computing multiple SPMs for a 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 5.3a. The construction is similar to 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-d 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 5.3b illustrates this process.

Algorithm 1 SELECTSTARTPOINT(i, q)

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

```
if $i = 0$ then
    return $p$
end if

$r \leftarrow$ LOCATE$(i - 1, q)$
switch ( $r$ )
    case $R1$ : $B$ : $C$ :
        return left$(s_{i-1})$
    case $R2$ : $A$ : left$(s)$ : right$(s)$ :
        return SELECTSTARTPOINT$(i - 1, q)$
    case $R3$ : $D$ : $E$ :
        return right$(s_{i-1})$
    case $R4$ :
        return SELECTSTARTPOINT$\big(i - 1,$ REFLECT$(q, s_{i-1})\big)$
end switch
```

5.2.3 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 and all of the SPMs. This produces a subdivision of s_n into a collection of $O(n)$ subsegments. Note that, due to our construction of the subsegments, each subsegment is fully contained in a single region of each SPM. For each subsegment we locate the largest i for which the subsegment is in either the left or right

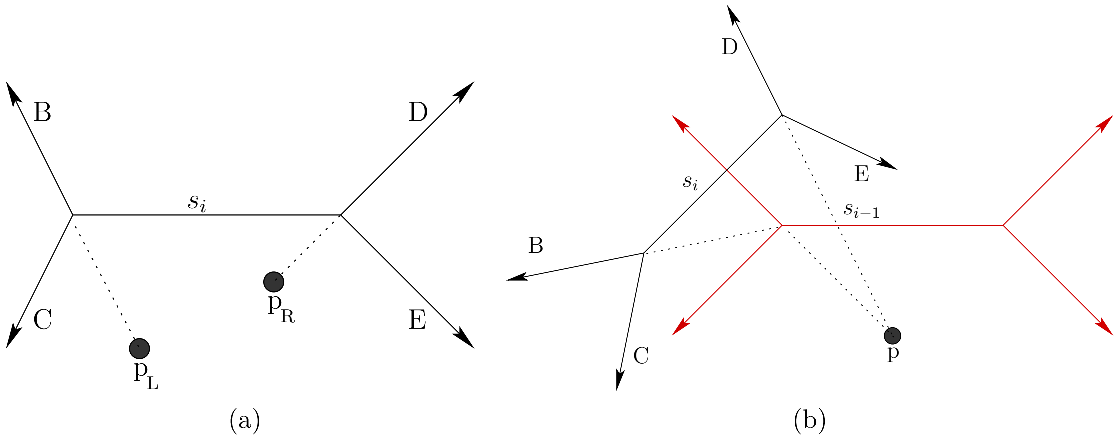


Figure 5.3: Computing the Shortest Path Map for segment s_i depends on the Shortest Path Map for segment s_{i-1}.

region of the SPM for s_i. Then we construct the complete path by executing ExtractPath$(i-1, \text{left}(s_i))$ or ExtractPath$(i-1, \text{right}(s_i))$ respectively, appended with the shortest direct path from that point to the subsegment, with appropriate reflections for bounce regions along the way from s_i to the subsegment of s_n. If there is no such i, the technique is similar, but uses the start point p instead, treating it as a degenerate segment. Pseudocode for the path extraction for each candidate can be found in Algorithm 2; the intuition is to traverse backward through the SPMs to p, adding a new edge to the path at each left, right, and bounce event. In this way, each subsegment generates a candidate path, and the ToS algorithm simply selects the shortest from among these candidate paths.

5.3 Algorithm Description

This section introduces our algorithm for optimal VBPE. We begin with a cell decomposition of the environment into conservative regions to compute the entire PEG (Section 5.3.1). We than provide a characterization of solution strategies (Section 5.3.2) before we introduce our algorithm. Starting from an empty sequence, we maintain a priority queue which stores sequences of PEG-nodes. The priority queue orders

Algorithm 2 ExtractPath(i, q)

```
Input: A SPM index $i$, and a query point $q$
 Data: A list $tour$ which stores points along our the optimal tour

if $i = 0$ then
    $tour$.insert($start_{pt}$)
    return $tour$
end if

$r \leftarrow$ Locate$\big(i, q\big)$
switch ( $r$ )
    case $R1$: $B$: $C$ :
        $tour \leftarrow$ ExtractPath$\big(i - 1, \mathrm{left}(s_i)\big)$
        return $tour$.insert$\big(\mathrm{left}(s_{i-1})\big)$
    case $R2$: $A$: left($s$): right($s$) :
        $tour \leftarrow$ ExtractPath$\big(i - 1, q\big)$
    case $R3$: $D$: $E$ :
        $tour \leftarrow$ ExtractPath$\big(i - 1, \mathrm{right}(s_i)\big)$
        return $tour$.insert$\big(\mathrm{right}(s_{i-1})\big)$
    case $R4$ :
        $reflect_{pt} \leftarrow$ Reflect$(q, s_i)$                ▷ reflect point across segment
        $tour \leftarrow$ ExtractPath$\big(i - 1, r\big)$

        ▷ calculate “bounce” point
        $bounce_{pt} \leftarrow$ LineIntersection$(s_i, (r, tour.\mathrm{back}())$
        return $tour$.insert($bounce_{pt}$)
end switch

return $tour$
```

the sequences by the length of the tour of segments. Using this priority queue, the algorithm performs a forward search to construct a complete solution strategy. Section 5.3.3 presents the details of this forward search. As the search progresses, it becomes necessary to perform a pruning operation. The details of the pruning operation(s) and an accompanying dominance relation appear in Section 5.3.4.

5.3.1 Cell Decomposition and Pursuit-Evasion Graph

We use the technique of Guibas, Latombe, LaValle, Lin, and Motwani [27], described in Chapter 4, to perform a cell decomposition of W into conservative regions. Atop this decomposition, we compute the PEG, which has one node for each unique sequence of gap labels at each conservative region.

5.3.2 Solution Sequences

In this section, we characterize solution strategies in terms of the sequences of conservative region boundary edges that are crossed. Using this characterization our algorithm will be able to discard many suboptimal sequences.

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

Theorem 1. *Let γ denote a solution strategy, and let $(s_1, \ldots, s_n)$ denote the sequence of conservative region boundary edges crossed by γ. Then any other pursuer trajectory γ' that crosses $(s_1, \ldots, s_n)$ in the same order without crossing any other boundaries is also a solution strategy.*

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

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

Note the pursuer can only follow such a segment sequence if each successive pair of segments lies on a single conservative region. Specifically, we require that for any

Algorithm 3 FORWARDSEARCH(p)

```
Input: a start point $p$, a pruning unary operator $prune$
Data: a priority queue $pq$ for sequences of PEG-nodes ordered by length of the ToS
Data: segment sequences are denoted as $\hat{s} = (s_1, \ldots, s_n)$

$pq$.insert(GETROOT($p$))            ▷ start with single contaminated node
while not $pq$.empty() do
    $\hat{s} \leftarrow pq$.top()            ▷ top sequence in the $pq$
    $s_n \leftarrow$ last($\hat{s}$)            ▷ PEG-node reached by following sequence
    if label($s_n$) $= 0 \cdots 0$ then            ▷ test for a solution
        return $\hat{s}$
    end if

    ▷ Outgoing directed edges from $PEG$-node
    for each $out$ in OUTGOINGNODES($s_n$) do
        $newseq \leftarrow$ APPENDTOSEQUENCE($\hat{s}$, $out$)            ▷ append sequence
        if not PRUNABLE($newSeq$) then
            $pq$.insert($newSeq$)            ▷ add new PEG-node sequence to $pq$
        end if
    end for
end while

return NO SOLUTION
```

sequence $(s_1, \ldots, s_n)$ stored at a PEG-node N, we have that s_i and s_{i+1} lie on the same conservative region, for each $i = 1, \ldots, n-1$, and s_n lies in the conservative region corresponding to the node N. We call such a sequence a *valid sequence* for N. If there exists a solution strategy that passes through a valid sequence, we call this sequence a *solution sequence*.

5.3.3 The Forward Search

Our planner to compute an optimal solution strategy uses a forward search approach [47]. We maintain a priority queue that stores sequences of PEG-nodes that correspond to a path through the PEG, ordered by the length of the ToS. Initially the priority queue is empty, but as we begin our search we can execute a query called

GetRoot that returns the PEG-node which corresponds to the pursuer's initial state. This node will have a shadow label that corresponds to being fully contaminated $(1 \cdots 1)$. We use the node returned by the GetRoot query as the root of our search (Algorithm 3 Line 1). Once our search is rooted (root node added to the priority queue) we can begin our search in earnest. At each iteration, the sequence $\hat{s} = (s_1, \ldots, s_n)$ at the head of the priority queue will correspond to a path from the root PEG-node to the current PEG-node s_n. New sequences are generated by iterating over all of s_n's outgoing directed edges (Algorithm 3 Line 10) and appending the target of the directed edge to the current sequence (Algorithm 3 Line 11). If the resulting sequence is not Prunable then it is added to the priority queue and the search continues.

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 sequence that reaches a PEG-node whose shadow label is "all clear" $(0 \cdots 0)$, then we know that no additional node expansions will generate a shorter solution strategy, so the search terminates successfully.

We know that the returned solution is an optimal solution strategy because the termination condition for the search behaves in such a way that once a PEG sequence which corresponds to a solution strategy appears at the top of the priority queue we return immediately. Since the priority queue is ordered according to the sequences ToS's we are assured that there does not exist another PEG sequence that also corresponds to a solution strategy whose tour is shorter. A detailed example that employs our algorithm appears in Appendix B.

Up to this point, we have been purposefully vague as to the implementation of the Prunable operation from Algorithm 3 Line 12. The next section goes into more detail about this operation and presents a dominance relation for sequences of

PEG-nodes that can be utilized to prune suboptimal paths during the forward search.

5.3.4 Pruning and Path Dominance

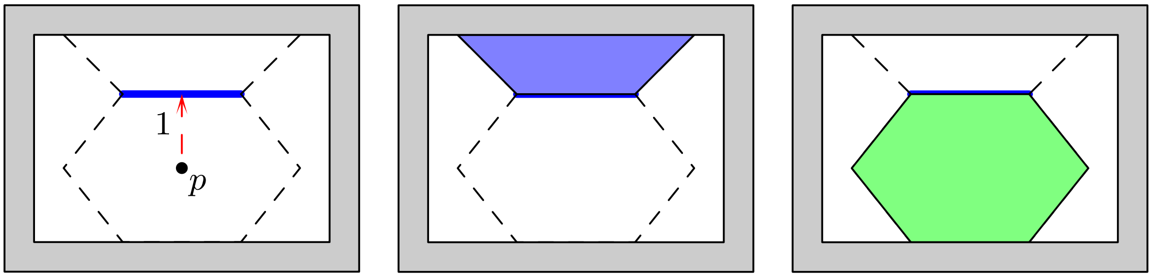
This section delves more deeply into the details surrounding the Prunable operation which determines whether a PEG sequence is added to our priority queue during the forward search. We discuss five different pruning strategies that can be used independently or in conjunction with one another to form the Prunable operation found in Algorithm 3.

Naïve approach – No pruning

The naïve approach to implementing the forward search is to ignore the concept of pruning altogether and to just add any PEG sequence to the priority queue. The UnavailingPruning pruning strategy which appears in Algorithm 4 behaves in this manner. However, this approach has the critical downfall of not progressing beyond the first conservative region boundary edge. The following claim and the accompanying discussion illustrate why this occurs.

Claim 1. *If* Prunable *is equivalent to* UnavailingPruning*, 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 q at point p. PEG-node q corresponds to a convex conservative region with k sides where $k \geq 3$. Without loss of generality we say that edge e_1 is a distance of 1 away from point p and edges $e_2 \ldots e_k$ are a distance of at least $1 + \epsilon$ away from p where $\epsilon > 0$, as seen in Figure 5.4. When the single element sequence q is expanded there are k new sequences added to the priority queue. The next sequence to appear at the front of the priority queue will be $\hat{r} = (q, r)$, where r is a PEG-node reached by



critical boundary Sequences of even length Sequences of odd length

Figure 5.4: The scenario that occurs when the UNAVAILINGPRUNING strategy is used. Initially the pursuer must travel to the closest critical boundary edge. Then for sequences of even length the pursuer will be in the blue conservative region. For sequences of odd length, the pursuer will be in the green conservative region.

Algorithm 4 UNAVAILINGPRUNING

Input: a sequence of PEG-nodes $S = \{s_1 \dots s_k\}$

```
function PRUNABLE($S$)
    return false
end function
```

travelling from q and crossing the critical boundary associated with e_1. This sequence has a cost of 1. Similar to node q, node r corresponds to a convex conservative region with j sides where $j \geq 3$. We already know that the cost of getting to node r 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 sequence will be generated $\hat{s} = (q, r, s)$ that has a cost of 1 which is smaller than all of the preexisting two-element sequences and the newly generated three-element sequences. This sequence is unique because the conservative region that corresponds to q is the same conservative region that corresponds to s. At this point it is fairly straightforward to show that at each successive iteration the sequence that appears at the top of the priority queue will reside either in the conservative region corresponding to q (odd length sequences) or the conservative region corresponding to r (even length sequences) and will have a cost of 1.

Algorithm 5 CycleCheckPruning

Input: a sequence of PEG-nodes $S = \{s_1 \dots s_k\}$

```
function PRUNABLE($S$)
    for each $s_i \in S, i \neq k$ do
        if label($s_i$) = label($s_k$) then
            return true
        end if
    end for
    return false
end function
```

Algorithm 6 RegressPruning

Input: a sequence of PEG-nodes $S = \{s_1 \dots s_k\}$

```
function PRUNABLE($S$)
    for each $s_i \in S, i \neq k$ do
        if label($s_i$) $\gg$ label($s_k$) then
            return true
        end if
    end for
    return false
end function
```

Minimal pruning

The previous section demonstrated the potential pitfall that can occur if PEG sequences aren't pruned. This section details some pruning strategies that ensures progress is made during the forward search. Recall the oscillatory behavior that was shown to occur when a pursuer reaches a critical boundary edge. The simple act of oscillating between a boundary segment is not in and of itself enough to prune a sequence as it may legitimately lead to previously unencountered PEG-nodes. However, we can show that after at most three consecutive crossings (Tables 5.1, 5.2, and 5.3) the sequence will begin to revisit PEG-nodes that have appeared earlier in the sequence. This essentially boils down to checking for a cycle within the PEG sequence. A pruning strategy that performs this task appears in Algorithm 5. This pruning

Table 5.1: Table that corresponds to a pursuer making repeated crossings over an Appear/Disappear event boundary. Scenarios 1 and 2 consider when a disappear event occurs first whereas Scenario 3 describes what occurs when an appear event occurs first.

	Scenario 1	Scenario 2	Scenario 3
	(Initially Contaminated)	(Initially Clear)	(Initially Empty)
Initial	1	0	empty
Action	Disappear	Disappear	Appear
Label	empty	empty	1
Action	Appear	Appear	Disappear
Label	0	0	empty
Action	Disappear		
Label	empty		

technique can be extended one step further by not only checking for cycles, but also checking to make sure the search is not regressing by visiting a PEG-node whose shadow label is dominated (Section 2.2.1) by a PEG-node that appears earlier in the sequence. A pruning strategy that performs this task appears in Algorithm 6, The general idea is that if by following a path that corresponds to the current PEG sequence we end up “losing” information, then a suboptimal decision on expansion was made at some previous point in the construction of the sequence.

Aggressive Pruning – Removing Suboptimal paths

Notice that there are an infinite number of PEG sequences which correspond to valid segment sequences. To account for this, we introduce a notion of sequence dominance that we use to construct more aggressive pruning strategies in order to expedite the forward search due to pruning suboptimal paths.

Table 5.2: Table that corresponds to a pursuer making repeated crossings over a Split/Merge event boundary. Scenarios 4-7 detail the various PEG-nodes that are visited when the merge event occurs first.

	Scenario 4	Scenario 5	Scenario 6	Scenario 7
	Clear	Clear	Contaminated	Contaminated
	Clear	Contaminated	Clear	Contaminated
Start	00	01	10	11
Action	Merge	Merge	Merge	Merge
Label	0	1	1	1
Action	Split	Split	Split	Split
Label	00	11	11	11
Action		Merge	Merge	
Label		1	1	

Table 5.3: Table that corresponds to a pursuer making repeated crossings over a Split/Merge event boundary. Scenarios 8 and 9 consider when a merge event occurs first.

	Scenario 8	Scenario 9
	(Initially Contaminated)	(Initially Clear)
Start	1	0
Action	Split	Split
Label	11	00
Action	Merge	Merge
Label	1	0

Definition 24. A segment sequence $\hat{r} = (r_1, \ldots, r_m)$ *dominates* a segment sequence $\hat{s} = (s_1, \ldots, s_n)$ if:

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

(b) For any segment sequence $\hat{a} = a_1, \ldots, a_k$ for which $(\hat{r}, \hat{a})$ and $(\hat{s}, \hat{a})$ are valid sequences, we have

$$\ell\Big(ToS(\hat{r}, \hat{a})\Big) \leq \ell\Big(ToS(\hat{s}, \hat{a})\Big) \ ,$$

in which ℓ denotes the length of a path.

(c) The shadow label of the PEG-node reached by $\hat{r}$ dominates the shadow label of the PEG-node reached by $\hat{s}$.

A conservative condition for stating that part (b) of Definition 24 is satisfied for segment sequences $\hat{r} = (r_1, \ldots, r_m)$ and $\hat{s} = (s_1, \ldots, s_n)$ is to show that

$$\ell\Big(ToS(\hat{r})\Big) + \textsc{MaxDistance}(r_m, s_n) \leq \ell\Big(ToS(\hat{s})\Big).$$

This condition is conservative because $\textsc{MaxDistance}(r_m, s_n)$ returns the maximum distance between two segments r_m and s_n, so the above inequality tests for worst-case behavior where the ToS for $\hat{r}$ arrives at r_m at a maximal distance from s_n which is the entry point for $\hat{s}$ into the conservative region. This pruning strategy appears in Algorithm 7.

Algorithm 7 ConservativePruning

```
Input: a sequence of PEG-nodes $S = \{s_1 \ldots s_k\}$
 Data: a data structure $nd$ that maintains a list of non-dominated sequences that
       reach a given conservative region

function PRUNABLE($S$)
    for each $R \in$ GETSEQUENCES($nd, s_k$) do                    ▷ $R = \{r_1 \ldots r_j\}$
        $cost_R \leftarrow$ TourOfSegments($R$)
        $dist \leftarrow$ MAXDISTANCE($r_j$, $s_k$)
        $cost_S \leftarrow$ TourOfSegments($S$)
        if label($r_j$) $\gg$ label($s_k$) then                     ▷ $R$ dominates $S$
            if $cost_R + dist < cost_S$ then       ▷ Path through $R$ is always preferable
                return true
            end if
        else if label($s_k$)$\underline{\gg}$ label($r_j$) then           ▷ $S$ strictly dominates $R$
            if $cost_S + dist < cost_R$ then       ▷ Path through $S$ is always preferable
                $nd$.remove($R$)              ▷ Remove $R$ from the "non-dominated" list
            end if
        end if
    end for
    return false
end function
```

A less conservative (and therefore more "Medial" in terms of pruning potential) condition for stating that part (b) of Definition 24 is satisfied for segment sequences $\hat{r} = (r_1, \ldots, r_m)$ and $\hat{s} = (s_1, \ldots, s_n)$ is to show that

$$\ell(ToS\big(\hat{r}, s_n, \hat{a})\big) \leq \ell\big(ToS(\hat{s}, \hat{a})\big).$$

which is equivalent to

$$\ell\Big(ToSToPoint(\hat{r}, \text{left}(s_n)\Big) \leq \ell\Big(ToS(\hat{s})\Big)$$

and

$$\ell\Big(ToSToPoint(\hat{r}, \text{right}(s_n)\Big) \leq \ell\Big(ToS(\hat{s})\Big).$$

The intuition is that by performing the TosToPoint subroutine at the endpoints of s_n we can find the farthest point on s from which to perform our shortest path query. This occurs because the farthest distance between a point and a line segment occurs at one of the endpoints. An implementation of this strategy appears in Algorithm 8.

A more Aggressive (in terms of pruning potential) strategy can be achieved by adhering to a more strict application of the condition stated in part (b) of Definition 24. Recall, that if part (a) of Definition 24 is satisfied then r_m and s_n are known to be boundary segments of the same conservative region. It follows that any future sequence of segments $\hat{a} = (a_1, \ldots, a_k)$ must begin in same conservative region that $\hat{r}$ and $\hat{s}$ terminate in. This effectively means that r_m, s_n, and a_1 are all boundary edges of the same conservative region. Therefore, we can use the following relations to satisfy the necessary condition in part (b) of Definition 24 to test for dominance between segment sequences $\hat{r} = (r_1, \ldots, r_m)$ and $\hat{s} = (s_1, \ldots, s_n)$

$$\ell\Big(ToSToPoint(\hat{r},\ \text{left}(a_1)\Big) \leq \ell\Big(ToSToPoint(\hat{s},\ \text{left}(a_1))\Big)$$

and

$$\ell\Big(ToSToPoint(\hat{r}, \text{right}(a_1)\Big) \leq \ell\Big(ToSToPoint(\hat{s}, \text{right}(a_1))\Big).$$

Algorithm 8 MedialPruning

Input: a sequence of PEG-nodes $S = \{s_1 \dots s_k\}$

Data: a data structure nd that maintains a list of non-dominated sequences that reach a given conservative region

```
function PRUNABLE($S$)
    for each $R \in$ GETSEQUENCES($nd$, $s_k$) do                    ▷ $R = \{r_1 \dots r_j\}$
        if label($r_j$) $\gg$ label($s_k$) then                          ▷ $R$ dominates $S$
            seg ← BoundarySegment($s_{k-1}$, $s_k$) ▷ segment between $s_{k-1}$ with $s_k$
            $rcost_{p1}$ ← TourOfSegmentsToPoint($R$, $seg.p1$)
            $rcost_{p2}$ ← TourOfSegmentsToPoint($R$, $seq.p2$)
            $cost_S$ ← TourOfSegments($S$)

            ▷ Path through $R$ is always preferable
            if $\big((rcost_{p1} < cost_S)$ and $(rcost_{p2} < cost_S)\big)$ then
                return true
            end if
        else if label($s_k$)$\underline{\gg}$ label($r_j$) then          ▷ $S$ strictly dominates $R$
            seg ← BoundarySegment($r_{j-1}$, $r_j$) ▷ segment between $r_{j-1}$ with $r_j$
            $scost_{p1}$ ← TourOfSegmentsToPoint($S$, $seg.p1$)
            $scost_{p2}$ ← TourOfSegmentsToPoint($S$, $seq.p2$)
            $cost_R$ ← TourOfSegments($R$)

            ▷ Path through $S$ is always preferable
            if $\big((scost_{p1} < cost_R)$ and $(scost_{p2} < cost_R)\big)$ then
                $nd$.remove($R$)              ▷ Remove $R$ from the "non-dominated" list
            end if
        end if
    end for
    return false
end function
```

The intuition is that by performing the TosToPoint subroutine on every potential future segment a, we can test to see if sequence $\hat{r}$ will always be preferred to sequence $\hat{s}$. Similar to the reasoning behind the Medial 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 $\hat{r}$ must have a shorter tour to both endpoints of any potential future segment a for it to dominate sequence $\hat{s}$. An implementation of this strategy appears in Algorithm 9.

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

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

As a result of this observation, our algorithm prunes any dominated sequence that is generated during the course of the search. The pruning operation is called when a new sequence is generated during node expansion. For a sequence to be added, it must not be dominated by any sequence belonging to a PEG-node that satisfies part (c) of Definition 24.

5.4 Results

In this section, we provide simulated results for our algorithm and compare them to the complete algorithm presented by GL^{3}M and a version of the GL^{3}M strategy that undergoes post-processing smoothing (Tour of Segments). We ran our simulations in

Algorithm 9 AggressivePruning

Input: a sequence of PEG-nodes $S = \{s_1 \ldots s_k\}$

Data: a data structure *nd* that maintains a list of non-dominated sequences that reach a given conservative region

```
function PRUNABLE($S$)
    for each $R \in$ GETSEQUENCES($nd$, $s_k$) do                              ▷ $R = \{r_1 \ldots r_j\}$
        if label($r_j$) $\gg$ label($s_k$) then                                   ▷ $R$ dominates $S$
            for each $seg \in$ CRBOUNDARYSEGMENTS($s_k$) do
                $rcost_{p1} \leftarrow$ TOUROFSEGMENTSTOPOINT($R$, $seg.p1$)
                $rcost_{p2} \leftarrow$ TOUROFSEGMENTSTOPOINT($R$, $seq.p2$)
                $scost_{p1} \leftarrow$ TOUROFSEGMENTSTOPOINT($S$, $seg.p1$)
                $scost_{p2} \leftarrow$ TOUROFSEGMENTSTOPOINT($S$, $seq.p2$)
                ▷ Path through $R$ is not preferable for all potential segments
                if $\big((rcost_{p1} > scost_{p1})$ or $(rcost_{p2} > scost_{p2})\big)$ then
                    Continue to next $R$
                end if
            end for
            ▷ Path through $R$ is always preferable
            return true
        else if label($s_k$)$\underline{\gg}$ label($r_j$) then                     ▷ $S$ strictly dominates $R$
            for each $seg \in$ CRBOUNDARYSEGMENTS($s_k$) do
                $rcost_{p1} \leftarrow$ TOUROFSEGMENTSTOPOINT($R$, $seg.p1$)
                $rcost_{p2} \leftarrow$ TOUROFSEGMENTSTOPOINT($R$, $seq.p2$)
                $scost_{p1} \leftarrow$ TOUROFSEGMENTSTOPOINT($S$, $seg.p1$)
                $scost_{p2} \leftarrow$ TOUROFSEGMENTSTOPOINT($S$, $seq.p2$)
                ▷ Path through $S$ is not preferable for all potential segments
                if $\big((scost_{p1} > rcost_{p1})$ or $(scost_{p2} > rcost_{p2})\big)$ then
                    Continue to next $R$
                end if
            end for
            ▷ Path through $S$ is always preferable
            $nd$.remove($R$)                          ▷ Remove $R$ from the “non-dominated” list
        end if
    end for
    return false
end function
```

three separate environments:

Figure 5.5	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.
Figure 5.6	This environment has 213 conservative regions, with a total of 26,620 PEG-nodes. The number of shadows per conservative region is at most 11.
Figure 5.7	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.

Table 5.4 shows the results of these simulations. A valuable resource that does not show up in Table 5.4 is computation time. The runtime of our simulations is dominated by the visibility cell decomposition and construction of the PEG. In our simulations the optimal strategy took slightly longer (handful of seconds) to generate the optimal strategy compared to the original GL^{3}M algorithm and the GL^{3}M algorithm with the Tour of Segments post-processing. Each of our simulations took less than a minute for cell decomposition, PEG construction, and the search.

For the environment that appears in Figure 5.5, the pursuer motion strategy returned by our algorithm bounces off of the rightmost diagonal and then heads toward the nook in the top-left corner. The motion strategy returned by the GL^{3}M algorithm gravitates towards the southern portion of the environment to escape the complex conservative region cells in the interior of the environment. There are a large amount of cells that have a visibility event along four boundary edges. There are a few larger cells under the tooth closest to the nook. These large cells with fewer transitions explain the difference between the two paths. The ToS subroutine improves upon the jaggedness that exists in the GL^{3}M strategy, but does not completely account for the difference that exists with the optimal path.

Table 5.4: The simulation results for the GL^{3}M algorithm, GL^{3}M with post-processing ToS path smoothing, and our optimal algorithm.

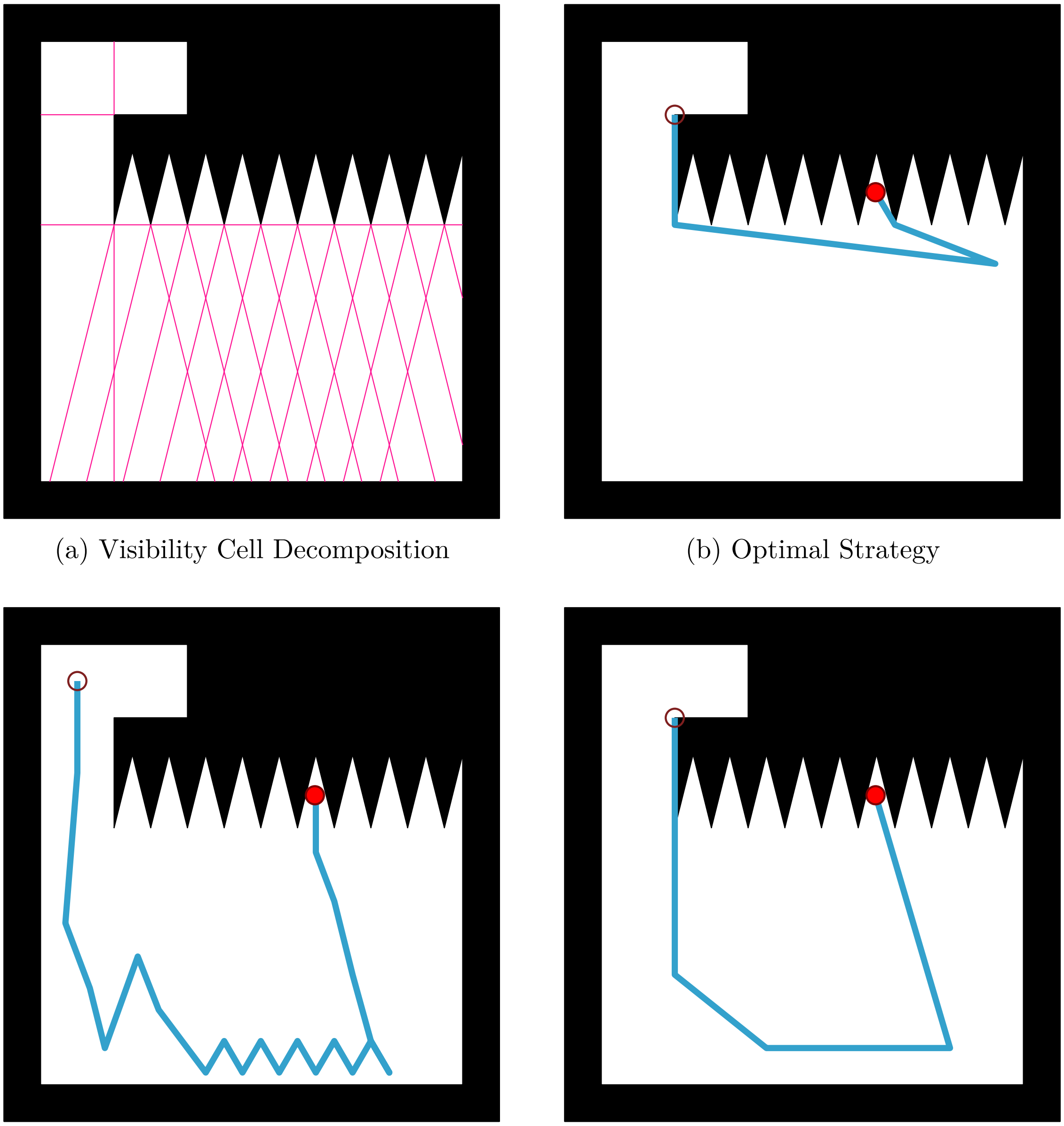
Environment	GL^{3}M path length (raw)	GL^{3}M path length (ToS)	Optimal path length
Figure 5.5	34.3705	22.3838	15.7671
Figure 5.6	47.9996	34.1259	31.3484
Figure 5.7	26.8714	24.066	17.8706

For the environment that appears in Figure 5.6, the pursuer motion strategy returned by our algorithm initially seems very similar to both the raw and smooth GL^{3}M strategies. However, in this case the Tour of Segments subroutine drastically improved the quality of the solution from the raw strategy. This environment has the largest disparity in runtime (8 seconds) between the GL^{3}M algorithm and our optimal strategy. It's apparent that the large number of very small conservative regions on the interior has an effect on the runtime of our algorithm. This leads to interesting questions about the ability to provide some kind of approximation guarantees with the benefit of decreased runtime at the expense of optimality.

The environment in Figure 5.7 demonstrates how poorly the GL^{3}M algorithm can perform as it goes about expanding according to a Breadth-First Search strategy. The post-processing smoothing subroutine does very little to improve the quality of the solution and it is apparent (triangle inequality) that the strategy returned by our algorithm is far superior to both incarnations of the GL^{3}M algorithm.

5.5 Concluding Remarks

While the algorithm presented by Guibas, Latombe, LaValle, Lin, and Motwani minimizes the number of PEG-nodes visited in a solution strategy, this is not a sufficient condition for generating optimal solution strategies. For comparison we compared our solution strategies against not only the solution strategies from the GL^{3}M algorithm, but also to the solution strategies from the GL^{3}M algorithm when applied with a post-processing step to compute the optimal strategy that visits the same sequence of conservative regions. Our simulation results clearly indicate that the optimal solution strategy is not necessarily a solution strategy that visits the fewest PEG-nodes. As mentioned above in Section 5.4 more work into approximation algorithms that relax the optimality guarantee but require fewer computation cycles is an excellent avenue for future work.



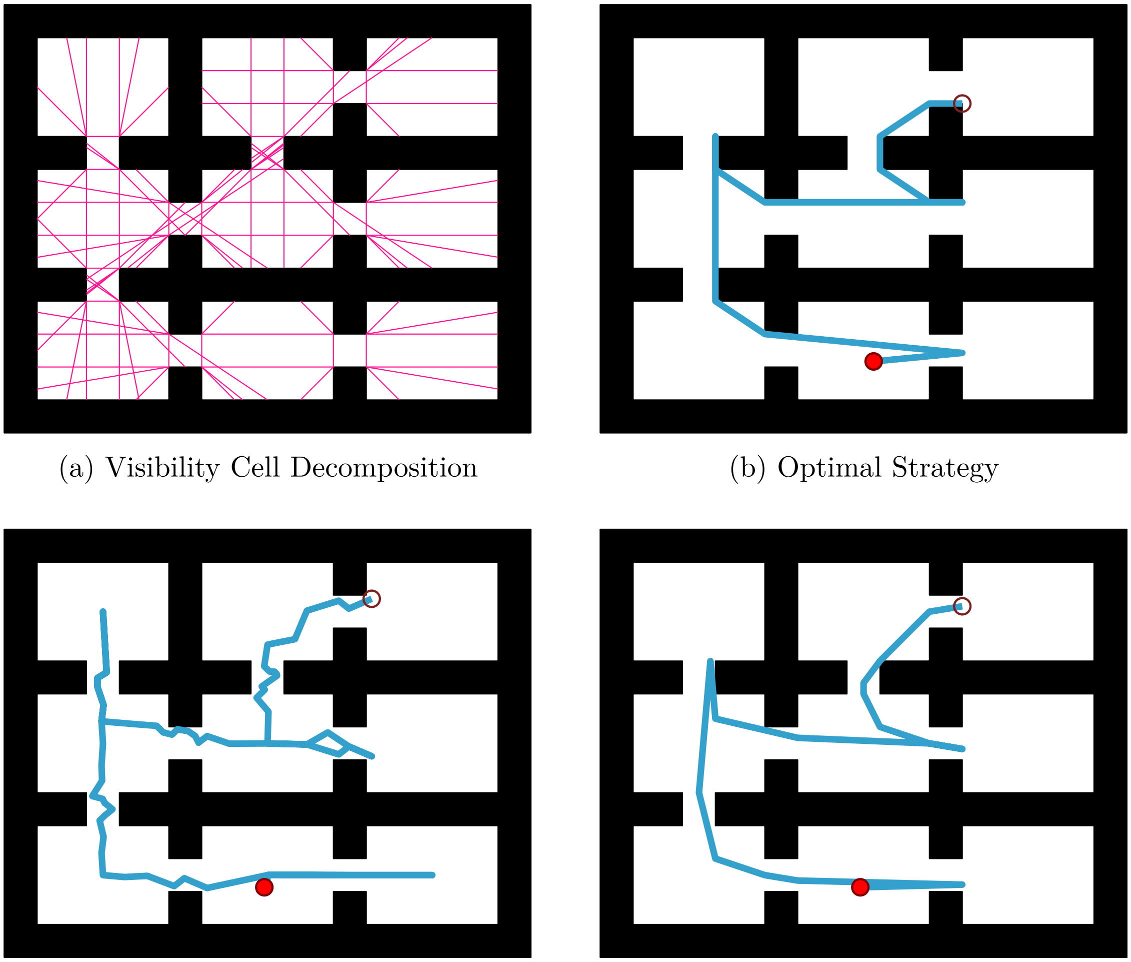
(a) Visibility Cell Decomposition

(b) Optimal Strategy

(c) GL^{3}M strategy

(d) ToS optimized GL^{3}M

Figure 5.5: An environment (5.5a) where the optimal pursuer strategy (5.5b) returned by our algorithm looks vastly different from both the original GL^{3}M strategy (5.5c) and the GL^{3}M strategy optimized using a ToS (5.5d).



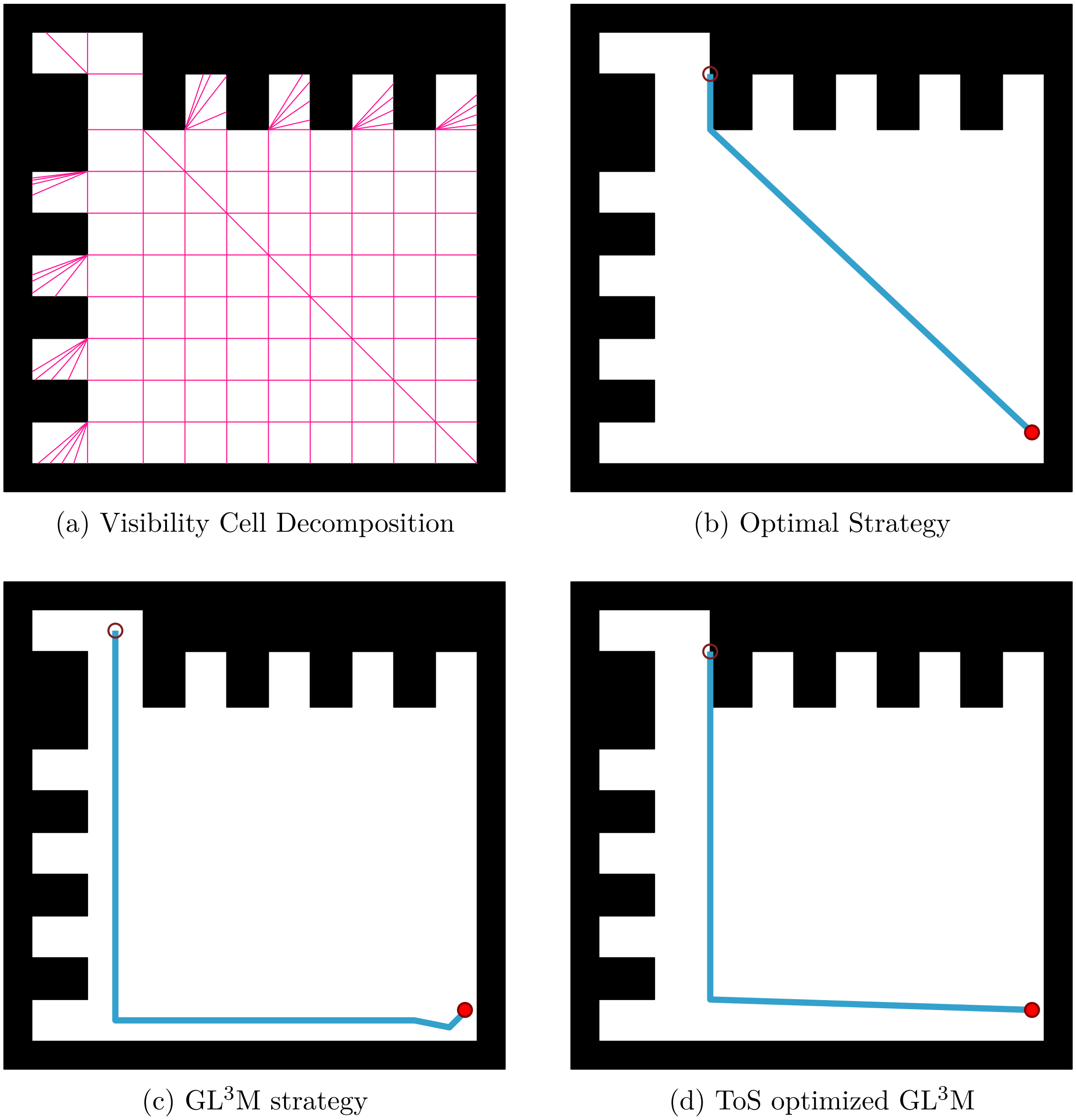
(a) Visibility Cell Decomposition

(b) Optimal Strategy

(c) GL^{3}M strategy

(d) ToS optimized GL^{3}M

Figure 5.6: An environment (5.6a) where the optimal pursuer strategy (5.6b) returned by our algorithm looks fairly similar to both the original GL^{3}M strategy (5.6c) and the GL^{3}M strategy optimized using a ToS (5.6d).



(a) Visibility Cell Decomposition

(b) Optimal Strategy

(c) GL^{3}M strategy

(d) ToS optimized GL^{3}M

Figure 5.7: An environment (5.7a) where the optimal pursuer strategy (5.7b) returned by our algorithm looks vastly different from both the original GL^{3}M strategy (5.7c) and the GL^{3}M strategy optimized using a ToS (5.7d).

Chapter 6

A Complete Algorithm for Multiple Pursuers

In this chapter we consider a variation on the visibility-based pursuit-evasion problem presented in [27] that utilizes a team of pursuers, as seen in Figure 6.1. The pursuers move through a polygonal environment seeking to locate an unknown number of evaders, each of which may move arbitrarily fast. The pursuers have an omni-directional field-of-view that extends to the environment boundary. The goal is to compute a joint strategy for the pursuers, or identify when such a strategy does not exist.

The main contribution of this work is a complete algorithm for multiple pursuer visibility-based pursuit-evasion that generates a solution strategy in the pursuers' joint configuration space. Our algorithm is a generalization of the previously-known complete algorithm for the case of a single pursuer (Chapter 4). Our algorithm identifies the different critical boundaries that occur when multiple pursuers are used during the search. Figure 6.1 demonstrates that a direct application of the decomposition used by GL^{3}M is insufficient because it does not account for the interaction amongst pursuers.

Similar to the GL^{3}M algorithm for a single pursuer, we start by decomposing the pursuers' joint configuration space into conservative regions. This decomposition of the pursuers' joint configuration space reduces the problem of finding a joint pursuer solution strategy to a discrete graph search. This decomposition is based on an

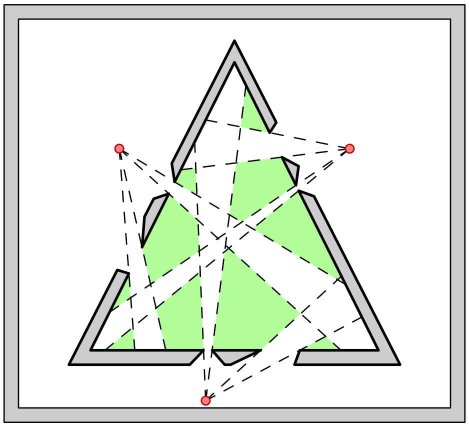


Figure 6.1: A configuration of three robots searching an environment. The shaded regions represent areas hidden to the pursuers.

analysis of the critical boundaries (Section 6.1) that occur when multiple pursuers are utilized in the search. We then provide an algorithm that uses Cylindrical Algebraic Decomposition (Appendix A) over these critical boundaries to produce a solution, or to conclude that no solution exists (Section 6.2).

A preliminary version of this work appears in [85].

6.1 Critical Boundaries

In this section, we provide a foundation for dividing W^n into conservative regions—within which shadow events cannot occur—by describing a complete set of critical boundaries at which such events *can* occur. Specifically, we examine the four different types of vertices that can compose the boundary of a shadow and establish critical boundaries where those vertices can change. The key idea is that each shadow can be characterized by its set of vertices, and that no shadow events can occur if the vertex set of every shadow region remains unchanged.

The vertices of every shadow can be classified into four types, as shown in Figure 6.2, which we call Types I, II, III, and IV.

- Type I vertices are environment vertices for which the adjacent edges in the

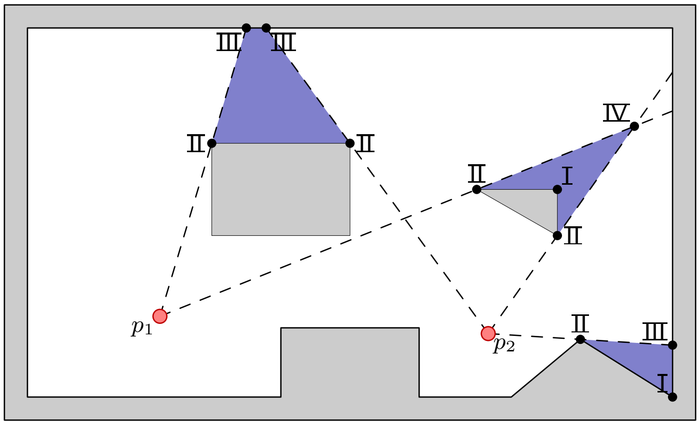


Figure 6.2: An environment with two pursuers illustrating the different types of shadow vertices.

shadow boundary lie along ∂W. Informally, these are vertices of the environment that no pursuer can see.

- Type II vertices are environment vertices, at which one of the two adjacent edges in the shadow boundary lies along ∂W and the other lies along an occlusion ray. Informally, these are vertices that are visible to some pursuer, but that block that pursuer's view of some other part of W.

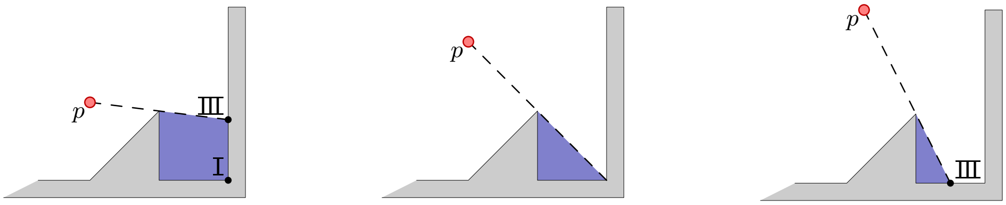
- Type III vertices are the endpoints of occlusion rays. Each lies on the interior of an edge of ∂W.

- Type IV vertices occur at intersections between occlusion rays.

We use the definition of conservative region from Section 4.2 to argue that just by thinking about when two shadow vertices can merge—and the inverse *split* events where a shadow vertex can split into two shadow vertices—we have identified all the ways in which a shadow can change. By definition, a region is conservative if it generates no shadow events, which means that the cardinality for the vertex set of the shadow stays the same. It follows that a shadow can only gain or lose shadow vertices when a pursuer crosses the boundary between conservative regions. By describing an exhaustive list of how two shadow vertices can merge at these critical boundaries, we

Table 6.1: The ten possible shadow vertex merges can be grouped into four general cases.

Event Types	Critical boundary occurs when...	Details
I-III, II-III, II-IV	pursuer colinear with two ∂W vertices	Sec. 6.1.1
III-III, III-IV	occlusion rays intersect on ∂W	Sec. 6.1.2
IV-IV	three occlusion rays share an intersection	Sec. 6.1.3
I-I, I-II, II-II, I-IV	never	Sec. 6.1.4



Before

At critical boundary

After

Figure 6.3: Type I and Type III vertices merge into a Type III vertex.

have identified all the ways in which a shadow can lose vertices. A inverse method of gaining vertices is the result of split events. Note that when a shadow has less than three shadow vertices, the shadow disappears. Likewise, a shadow appears when there are at least three shadow vertices.

The next step is to characterize the sets of joint configurations at which such vertex merges can occur. Considering all pairs of vertex types, there are ten distinct possible types of merges. We'll consider each of these ten cases. Fortunately, the ten cases can be grouped into four general categories that can be analyzed in similar ways. Table 6.1 summarizes the merge types.

6.1.1 Merges Resulting from Pursuers Colinear with a Pair of Environment Vertices

First, we argue that merge types I-III, II-III, and II-IV occur only when some pursuer is colinear with some pair of environment vertices.

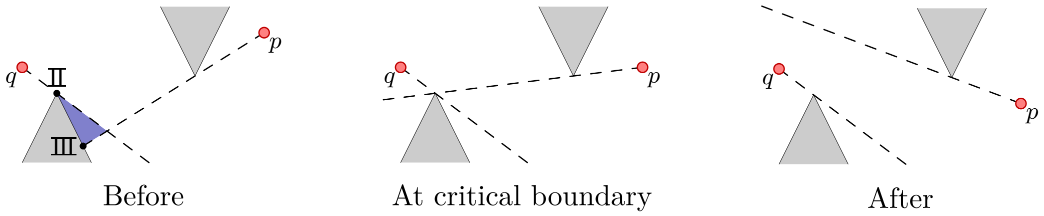


Figure 6.4: A Type II vertex merges with a Type III vertex, eliminating the shadow.

I-III merges

Consider the case in which a Type I and Type III vertex merge. This situation requires a vertex of ∂W to be coincident with the endpoint of an occlusion ray ∂W. Figure 6.3 shows how this can occur. On one side of this boundary, the shadow has a Type I vertex adjacent to a Type III vertex; on the other side, those vertices are replaced with a single Type III vertex.

Specifically, for a Type I vertex at $u = (x_\mathrm{u}, y_\mathrm{u})$ and a Type III vertex owned by pursuer $p = (x_\mathrm{p}, y_\mathrm{p})$ and induced by occlusion vertex $v = (x_\mathrm{v}, y_\mathrm{v})$, this kind of event occurs when

$$\begin{vmatrix} x_\mathrm{p} & y_\mathrm{p} & 1 \\ x_\mathrm{u} & y_\mathrm{u} & 1 \\ x_\mathrm{v} & y_\mathrm{v} & 1 \end{vmatrix} = 0. \tag{6.1}$$

Treating x_u, y_u, x_v, and y_v as constants, this equation expands to a polynomial of degree 1 in the variables x_p and y_p. To form the complete set of critical boundaries of this type, we must iterate over all n choices of pursuers, and all $\binom{m}{2}$ choices for $u = (x_\mathrm{u}, y_\mathrm{u})$ and $v = (x_\mathrm{v}, y_\mathrm{v})$.

II-III merges

For a Type II vertex to merge with a Type III vertex, we must have an occlusion ray of one pursuer colinear with an occluding vertex of another pursuer, as illustrated in Figure 6.4. This requires a pursuer p to be colinear with the two occluding vertices

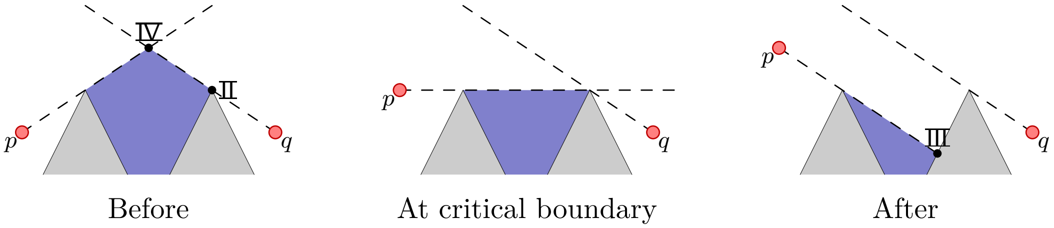


Figure 6.5: A Type II vertex merges with a Type IV vertex, creating a Type III vertex.

$u = (x_{\mathrm{u}}, y_{\mathrm{u}})$ and $v = (x_{\mathrm{v}}, y_{\mathrm{v}})$. Thus, the critical boundary polynomial is identical to Equation 6.1; the only difference is that, in this case, both $u = (x_{\mathrm{u}}, y_{\mathrm{u}})$ and $v = (x_{\mathrm{v}}, y_{\mathrm{v}})$ must be reflex (i.e. non-convex) vertices.

II-IV merges

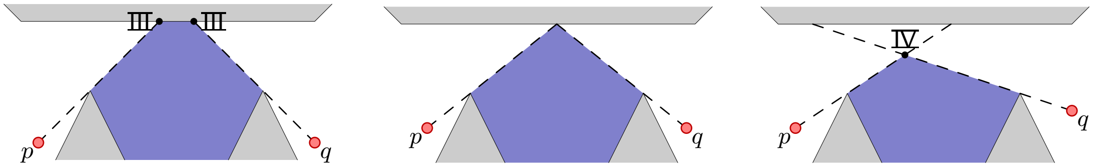
Likewise, for a Type II vertex to merge with a Type IV vertex, two occlusion rays from two different pursuers must intersect at the occluding vertex of one of those rays. See Figure 6.5. As in the previous two cases, this can occur only when a pursuer p is colinear with two vertices $u = (x_{\mathrm{u}}, y_{\mathrm{u}})$ and $v = (x_{\mathrm{v}}, y_{\mathrm{v}})$ of ∂W, and Equation 6.1 defines the critical boundary.

Number of Polynomials

For a fixed pursuer, the total number of critical event polynomials for these three merge types is at most $\binom{m}{2}$, yielding a maximum of $\binom{n}{1}\binom{m}{2}$ polynomials across all n pursuers.

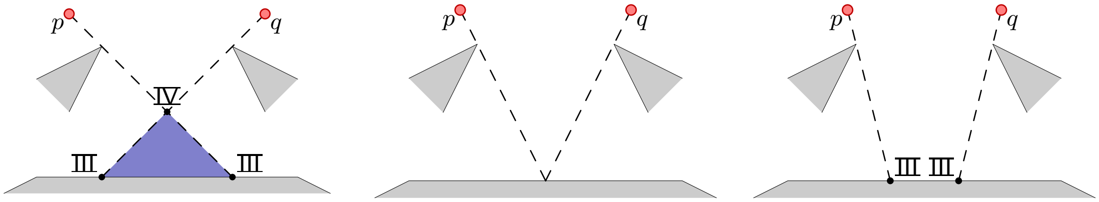
6.1.2 Merges Resulting from Two Occlusion Rays Intersecting on ∂W

Next we consider merge types III-III and III-IV, and argue that these events occur when occlusion rays from two distinct pursuers meet precisely on the environment boundary.



Before At critical boundary After

Figure 6.6: A Type III vertex merges with a Type III vertex creating a Type IV vertex.



Before At critical boundary After

Figure 6.7: A Type III vertex merges with a Type IV vertex, creating a Type III vertex.

III-III merges

For a Type III vertex to merge with another Type III vertex, these two vertices must occupy the same location along an edge of ∂W. Let p and q denote the pursuers that own these two vertices, and let u and v denote the respective occlusion vertices that generate the two Type III vertices. Finally, let w and z denote the endpoints of the environment edge on which the two Type III vertices lie. Figure 6.6 illustrates this situation.

These two vertices merge when the lines $\overleftrightarrow{pu}$, $\overleftrightarrow{qv}$, and $\overleftrightarrow{wz}$ all share an intersection point. This triple intersection occurs when

$$\begin{vmatrix} y_{\mathrm{u}} - y_{\mathrm{p}} & x_{\mathrm{p}} - x_{\mathrm{u}} & x_{\mathrm{u}}y_{\mathrm{p}} - x_{\mathrm{p}}y_{\mathrm{u}} \\ y_{\mathrm{v}} - y_{\mathrm{q}} & x_{\mathrm{q}} - x_{\mathrm{v}} & x_{\mathrm{v}}y_{\mathrm{q}} - x_{\mathrm{q}}y_{\mathrm{v}} \\ y_{\mathrm{z}} - y_{\mathrm{w}} & x_{\mathrm{w}} - x_{\mathrm{z}} & x_{\mathrm{z}}y_{\mathrm{w}} - x_{\mathrm{w}}y_{\mathrm{z}} \end{vmatrix} = 0. \tag{6.2}$$

The equation expands to a polynomial of degree 2 in four variables—namely x_{p}, y_{p}, x_{q}, and y_{q}—and 8 constants.

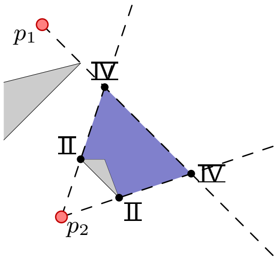


Figure 6.8: A Type IV vertex merging with a Type IV vertex with 2-robots.

III-IV merges

For a Type III vertex to merge with a Type IV vertex, again we need two occlusion rays to meet on ∂W. This situation is the same as the III-III case above, except that we are approaching from the opposite side; see Figure 6.7. As with the III-III case, this requires three lines (two occlusion rays and one environment edge) to meet a single point. As a result, Equation 6.2 describes the III-IV critical boundary as well.

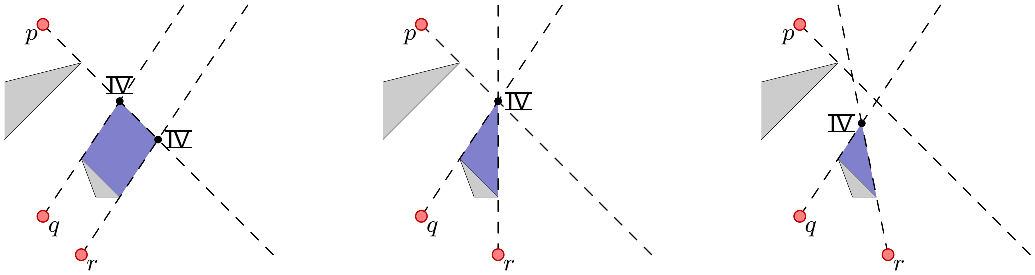
Number of Polynomials

These types of critical boundaries are defined by a pair of mutually visible environment vertices, along with an additional environment boundary edge. Therefore, for a fixed pair of pursuers, it can be instantiated at most $\binom{m}{3}$ different ways. It also depends on the positions of two different pursuers, of which there are $\binom{n}{2}$ unique combinations. Therefore, in total—across both III-III and III-IV—this type of critical boundary yields a maximum of $\binom{n}{2}\binom{m}{3}$ polynomials.

6.1.3 Merges Resulting from Three Occlusion Rays Meeting a Single Point

The final plausible merge type we consider is IV-IV. For two Type IV vertices to meet, we must have at least three occlusion rays that share a single intersection point.

Claim 2. *Two pursuers are not sufficient to produce a IV-IV merge event.*



Before At critical boundary After

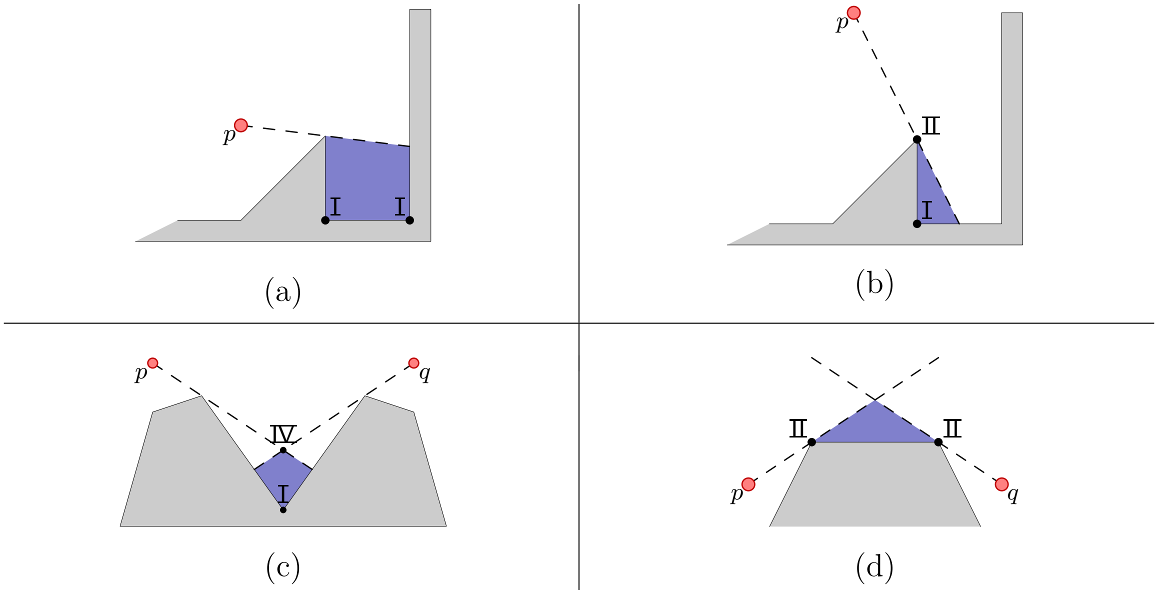
Figure 6.9: A Type IV vertex merging with a Type IV vertex requires multiple robots and creates a single Type IV vertex.

Proof. Assume that two-pursuers are sufficient for a IV-IV merge to occur, as illustrated in Figure 6.8. For two Type IV vertices to be adjacent, the shadow edge connecting them must be part of an occlusion ray, and without loss of generality we say that pursuer p_1 owns that occlusion ray. By definition a Type IV vertex has edges that are occlusion rays. So p_2 is the owner of two occlusion rays that intersect with the occlusion ray of p_1 creating two Type IV vertices. For a merge event to occur the three occlusion rays must be concurrent. However the two distinct occlusion rays originating from p_2 intersect only at p_2 and nowhere else. Thus two pursuers are incapable of causing a IV-IV merge.

Thus, a IV-IV merge can occur when three distinct pursuers have occlusion rays that meet at a single point (Figure 6.9). This is, in principle, similar to the situation from Section 6.1.2, except that the pursuers' movements can move all three relevant lines (occlusion vertices for p, q, and r are denoted by u, v, and w respectively):

$$\begin{vmatrix} y_\mathrm{u} - y_\mathrm{p} & x_\mathrm{p} - x_\mathrm{u} & x_\mathrm{u} y_\mathrm{p} - x_\mathrm{p} y_\mathrm{u} \\ y_\mathrm{v} - y_\mathrm{q} & x_\mathrm{q} - x_\mathrm{v} & x_\mathrm{v} y_\mathrm{q} - x_\mathrm{q} y_\mathrm{v} \\ y_\mathrm{w} - y_\mathrm{r} & x_\mathrm{r} - x_\mathrm{w} & x_\mathrm{w} y_\mathrm{r} - x_\mathrm{r} y_\mathrm{w} \end{vmatrix} = 0.$$

In this equation, the x and y coordinates for each of the three relevant pursuers form 6 total variables, and the coordinates of their three occlusion vertices form 6 constants. The expanded polynomial has degree 3.



(a)

(b)

(c)

(d)

Figure 6.10: Merge events that never occur: (a) I-I (b) I-II (c) I-IV (d) II-II.

This scenario requires three unique environment vertices to induce occlusion rays from the pursuers, there are at most $\binom{m}{3}$ places where this can occur. This type of merge also requires three pursuers and there are $\binom{n}{3}$ unique combinations of pursuers. In total this critical boundary yields a maximum of $\binom{n}{3}\binom{m}{3}$ polynomials.

6.1.4 Merges That Never Occur

Finally, we argue that the remaining four merge types can never occur.

- Merges that involve only environment vertices—that is, merges of types I-I, I-II, and II-II—cannot occur because environment vertices do not move, and therefore never merge with one another.

- Merges of type I-IV cannot occur because Type I and Type IV vertices are never adjacent. Notice that, in a shadow polygon, a Type I vertex is incident to two edges along ∂W, whereas a Type IV vertex is incident to two edges in the interior of W. Therefore, there always exists at least one other vertex between any Type I and Type IV pair.

Because these merges cannot occur, they do not generate any critical boundary polynomials.

6.2 Algorithm

Armed with this complete description of the critical boundaries in W^n, we can finally describe our algorithm for multiple-pursuer visibility-based pursuit-evasion in detail. The basic process is to use the critical boundaries to form a partition of W^n into conservative regions, to compute an adjacency graph of the full-dimensional cells in that partition, and then to search for a sequence of adjacent conservative regions that causes all of the shadows to be cleared.

6.2.1 Partitioning W^n via Cylindrical Algebraic Decomposition

The first step of our algorithm is to compute each of the critical boundary polynomials described in Section 6.1. This results in a collection $\mathcal{P}$ of $\mathrm{O}\left(n^3m^3\right)$ polynomials in the $2n$ variables $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$. Each of these polynomials can be constructed in constant time, so this step takes time $\mathrm{O}\left(n^3m^3\right)$.

We then use these polynomials as input to the standard cylindrical algebraic decomposition (CAD) algorithm [18], which generates a partition of $\mathbb{R}^{2n}$ into cells with dimensions ranging from 0 to $2n$. The CAD algorithm guarantees that, within each cell of the decomposition, the sign of each polynomial in $\mathcal{P}$ remains constant. In particular, because $\mathcal{P}$ includes every critical boundary curve, this implies that every cell of dimension $2n$ is either a conservative region of the joint free space, or an obstacle portion of the joint configuration space.

Moreover, each cell of dimension $2n - 1$ separates a pair of adjacent cells of dimension $2n$. Each $(2n-1)$-cell *may* correspond to a shadow event, but may also

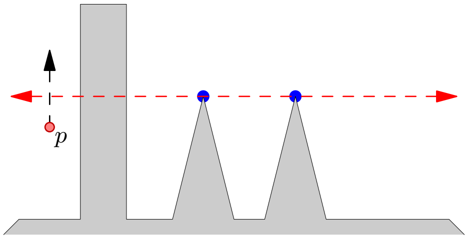


Figure 6.11: An example of a critical boundary(bitangent) polynomial passing through obstacles. Because the pursuer motion shown crosses this boundary, it moves to a new CAD cell, even though no shadow event occurs.

exist because of the CAD algorithm's need to form cells that are cylindrical, or may occur due to extensions of the critical boundaries—which, in the CAD algorithm, are treated as polynomials that do not stop at the environment boundary—beyond the portion of the free space in which they are relevant. See Figure 6.11.

6.2.2 Computing the Adjacency Graph of the Conservative Regions

Next, our algorithm forms an *adjacency graph* describing how the pursuers can move through those conservative regions.

- Each vertex of the adjacency graph corresponds to a $2n$-dimensional cell of the CAD within the joint free space.
- Edges of the adjacency graph correspond to $2n-1$-dimensional CAD cells, and connect vertices corresponding to conservative regions that share a portion of their boundaries.

There are two different approaches to the construction and search of the adjacency graph. The first [2, 46] has a multiply-exponential dependence on $2n$, whereas the second [76] takes doubly-exponential time in $2n$. The exact construction and search of the adjacency graph is beyond the scope of this dissertation, and the authors refer the reader to the original text.

In addition, we label each edge of the adjacency graph with the shadow events, if any, that occur when the pursuers move between the corresponding conservative regions. By examining the shadows before and after we can retroactively assign labels to $2n - 1$ cells that represent critical boundaries.

6.2.3 Path Generation

Finally, we can use the adjacency graph to search for a solution strategy for the pursuers. The intuition is to search through the Multi-Pursuer Pursuit-Evasion Graph (MP-PEG) induced by the adjacency graph.

1. Specifically, given a vertex v of the adjacency graph, let $k(v)$ denote the number of shadows that exist when the pursuers are within the conservative region corresponding to v. The MP-PEG contains $2^{k(v)}$ vertices for each adjacency graph vertex v. Each such vertex is labeled with a unique binary string of length $k(v)$, representing one possible combination of clear and contaminated shadow labels. The total number of MP-PEG vertices is $\sum_v 2^{k(v)}$.

2. A pair of MP-PEG vertices (u, v) is connected by a directed edge $u \to v$ if

   a) the adjacency graph vertices underlying u and v are connected in that graph, and

   b) the changes in shadow labels between u and v are correct, according to the rules introduced in Section 2.2.

The intuition is that each vertex of the MP-PEG fully describes one discrete information state that the pursuers might reach—including both their positions and the clear/contaminated status of each shadow—and that the edges represent “actions” that the pursuers can take to change those shadow labels.

Therefore, the final step of the algorithm is a forward search through the MP-PEG. The search starts from the pursuers’ initial position with all of the shadows labeled as contaminated, and terminates at a MP-PEG vertex with all of the shadows are labeled clear.

The forward search is done using a Breadth-first search(BFS) algorithm. The search takes time $\mathrm{O}\left(V + E\right)$ where V is the number of vertices in the MP-PEG and E is the number of edges. Since the MP-PEG is induced by the adjacency graph, any sequence of visited MP-PEG nodes can be mapped back to the original CAD, and the process of generating a continuous path is similar to extracting a path from the original CAD as done in the standard Schwartz and Sharir algorithm [76]. If the search fails to find a path, we know that a solution does not exist because BFS performs an exhaustive search. Since by definition a MP-PEG vertex describes one discrete information state that the pursuers might reach, the union of all MP-PEG vertices completely describes all possible information states for the pursuers. By conducting an exhaustive search of MP-PEG without finding a solution we conclude that there is no possible sequence of actions that the pursuers can take through the joint configuration space that guarantees the capture of the evader.

6.2.4 Algorithm Analysis

We begin the analysis of our algorithm by examining the individual steps of the algorithm. The dimension of the joint configuration space is $2n$. The number of polynomials in $\mathcal{P}$—which is used as input into the CAD algorithm—is the sum of the critical boundaries and is $\mathrm{O}\left(n^3m^3\right)$. The maximum degree among the polynomials in $\mathcal{P}$ is 3 (which occurs for the IV-IV merge event.)

The total running time [47] for the construction and adjacency test on our CAD is bounded by $(3 \cdot n^3m^3)^{\mathrm{O}(1)^n}$ where $\mathrm{O}\left(\cdot\right)$ means that there exists $c \in [0, \infty]$ such that the running time is bounded by $(3 \cdot n^3m^3)^{c^n}$ [47]. The number of cells [18,76] produced by our CAD is $\mathrm{O}\left(6^{6n+1} \cdot (n^3m^3)^{4n}\right)$. The running time for the entire algorithm is dominated by the the construction and adjacency test on the CAD.

Chapter 7

A Sampling Based Algorithm for Multiple Pursuers

Motivated by the complexity of the complete algorithm for multiple pursuer visibility-based pursuit-evasion in Chapter 6, this chapter presents a more practical solution. Once again we have a team of pursuers in a polygonal environment seeking to locate an unknown number of evaders, each of which may move arbitrarily fast. The pursuers have an omni-directional field-of-view that extends to the environment boundary. The goal is to compute a joint strategy for the pursuers, or identify when such a strategy does not exist.

The main contribution of this work is a probabilistically complete algorithm for multiple pursuer visibility-based pursuit-evasion that generates a solution strategy for the pursuers to execute (Figure 7.1) through the joint configuration space. Our algorithm creates a graph that maintains the pursuers' information state, and utilizes a sample generator that we treat as a "black box" to reason about unexplored areas in the pursuers' joint configuration space. Our algorithm has some similarity to the Probabilistic Road Map (PRM) algorithm [39], but differs in that our algorithm maintains information concerning the areas of the environment where the evader might be. The need for this additional information complicates both the update operations for the graph and the selection of samples.

This remainder of this chapter is structured as follows. Section 7.1 introduces a data structure that maintains a representation of the pursuers' joint information

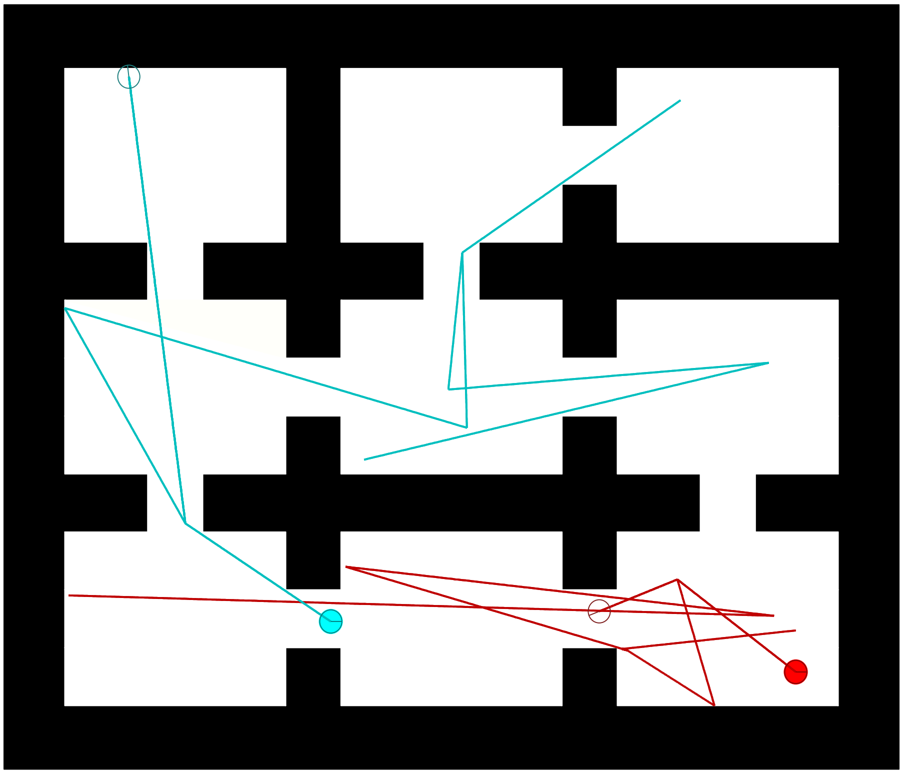


Figure 7.1: A pursuer strategy generated by our algorithm. Filled circles represent the pursuers' initial positions and open circles represent their goal positions.

state. Section 7.2 presents an algorithm that uses the aforementioned data structure to search for a pursuer solution strategy. Simulation results appear in Section 7.3 that show our algorithms ability to generate solution strategies for various sample generators.

A preliminary version of this work appears in [86].

7.1 Sample-Generated Pursuit-Evasion Graph

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

7.1.1 Graph Structure

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

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

For an edge to exist between any two vertices in the SG-PEG there must be a line segment in W^n that connects the joint pursuer configuration at the source vertex with the joint pursuer configuration at the target vertex. Given an arc of the SG-PEG, $e = (x, y)$, the edge stores a mapping from the reachable shadow labels in x to the corresponding shadow labels in y. Figure 7.2 offers a snapshot of how the SG-PEG tracks potential evader positions between configurations (dashed lines).

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

7.1.2 Edge Creation

When a new connection is established between a source and target vertex in the SG-PEG, the source's reachable shadow labels are used to update the target's reachable labels (Algorithm 10). In this section we discuss the shadow label update criterion, the update label subroutine, and the process of adding a new reachable label to a vertex.

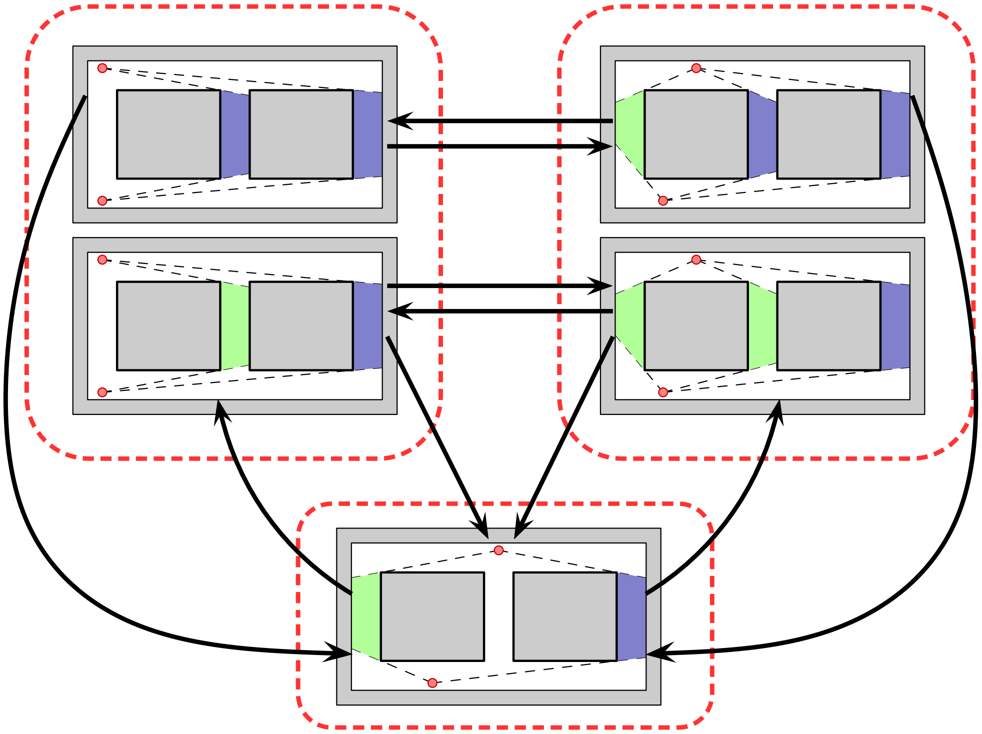


Figure 7.2: A snapshot of the SG-PEG. Dashed red lines indicate a unique joint pursuer configuration.

Computing a New Label

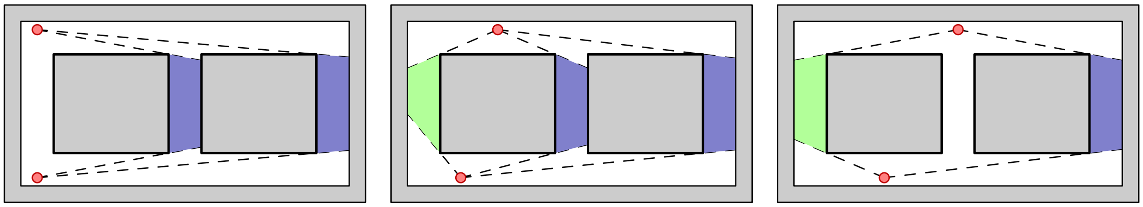
In Section 6.1, we provided a family of polynomials that identify the critical boundaries that indicate a change in a shadow's composition. Although complete, the quantity and complexity of the polynomials in this family makes the task of analytically identifying where these changes occur computationally expensive. Instead, we update the shadow labels numerically.

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

Algorithm 10 ADDEDGE(s, t)

Input: a source vertex s and a target vertex t

```
for each label in $s$'s reachable set do
    newlabel ← UPDATE($s$.jpc, label, $t$.jpc)
    ADDREACHABLE($t$, newlabel)
end for
```



Before During After

Figure 7.3: An illustration of the update step. Initially there are two contaminated shadows (purple). During the UPDATE a new label appears. At the conclusion of the UPDATE method, there are two shadows: a cleared shadow (green) and a contaminated shadow (purple).

The computation of a new shadow label (Algorithm 11) takes as input two joint pursuer configurations, a source and target, and a shadow label corresponding to the shadow region at the source configuration. The output is the shadow label that results from the pursuers moving from the source configuration to the target configuration given the initial shadow label. Figure 7.3 illustrates this process. Initially, there are two contaminated shadows. As the pursuers move to the target configuration, a shadow appears as the pursuers move to the right (a cleared shadow). As the pursuers reach the target configuration, the central shadow disappears.

We begin by partitioning (Algorithm 11 line 2) the segment connecting the source and target configurations in F^n into a finite collection of evenly spaced joint pursuer configurations. We then loop through this collection of joint pursuer configurations, updating the shadow label along the way, returning the final label of the sequence.

The process of computing the new shadow labels for our discretized segments appears in Algorithm 11 lines 5-15. The process starts by computing the shadow

Algorithm 11 COMPUTELABEL(p, l, p')

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

Output: the label that results when travelling from p to p' starting from label l

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

for each $p_i$, $p_{i+1}$ where $i < k$ do
    oldshadows $\leftarrow$ SHADOWREGION($p$)
    newshadows $\leftarrow$ SHADOWREGION($p'$)
    newlabel $\leftarrow 0 \cdots 0$                ▷ initially all cleared
    for each $s'$ in newshadows do
        for each $s$ in oldshadows do
            if label$_s = 1$ and $s'$ intersects $s$ then
                newlabel$_{s'} \leftarrow 1$
            end if
        end for
    end for
    label $\leftarrow$ newlabel
end for
return label
```

regions of both the source and target configurations. We initialize the label corresponding to the target configuration as all cleared. We check all of the shadows in the shadow region of the goal configuration for an intersection with contaminated shadows belonging to the shadow region of the source configuration. If an intersection with a contaminated shadow occurs then the corresponding shadow in the target configuration is also labelled as contaminated.

Adding a Reachable Label

The final step involves adding the newly computed shadow label to the target vertex (Algorithm 12). It may also be the case that the individual shadows of the new label are all "cleared", in which case a solution has been found. If the target vertex contains a shadow label in its set of reachable labels that dominates the new shadow

Algorithm 12 AddReachable(v, l)

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

```
if $v$ contains a label that dominates $l$ then return
end if

add $l$ to $v$ as a reachable label
delete labels in $v$ dominated by $l$
if AllClear($l$) then
    Output Solution $v$                                   ▷ Is $l$ a solution?
end if

for each out in Neighbors($v$) do
    newlabel ← ComputeLabel($v$.jpc, $l$, out.jpc)
    AddReachable(out, newlabel)
end for
```

label, then the new label does not contribute any new information and we return. Similarly, if there are labels in the vertex's set of reachable labels that are dominated by the new shadow label, then those labels are removed. If the new shadow label is not dominated and is not a solution strategy then we add the new shadow label to the vertex's reachable set. This label now permeates the graph recursively via the vertex's outgoing edges. A label is calculated for each of the vertex's neighbors, and if this label is added to the neighbors reachable set, then the process repeats itself. The process ends when no additional reachable labels are found.

Note that if a vertex does not belong to the same connected component as the *root* vertex then its set of reachable labels is empty. Because of the recursive nature of Algorithm 12, a vertex that serves as a bridge between the connected component containing the *root* vertex and another connected component will cause the reachable data to permeate through the SG-PEG.

Algorithm 13 SOLVE(p, W, A)

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

```
 1: AddVertex($p, \{0 \cdots 0\}$)
 2: while a solution has not been found do
 3:     $s \leftarrow A$.GetSample()
 4:     $x \leftarrow$ AddVertex($s$)
 5:
 6:     for each $y$ in SG-PEG vertices do
 7:         if ($\overline{xy} \subset W^n$) and
              length($x, y$) < maxlength and
              cycleLength($x, y$) > mincycle then
 8:             AddEdge($x, y$)                          ▷ Digraph edge
 9:             AddEdge($y, x$)                          ▷ Digraph edge
10:         end if
11:     end for
12: end while
13: return ExtractSolution(solution)
```

7.2 Algorithm

In this section we detail how our algorithm uses a SG-PEG to search for a pursuer solution strategy. Our algorithm (Algorithm 13) begins by creating a SG-PEG vertex. This vertex's joint pursuer configuration is the initial joint pursuer configuration supplied to our algorithm and it's set of reachable shadow labels contains only a single label whose shadows are all contaminated. This is the *root* vertex of our SG-PEG. We then proceed by obtaining samples in W^n, checking these samples for potential connections with existing vertices in the SG-PEG graph, and update the SG-PEG where necessary when edges are created.

7.2.1 Abstract Sampler

Our main search algorithm uses an *abstract sampler* to return a joint pursuer configuration (Algorithm 13 line 3).

Definition 25. An *abstract sampler* is a joint probability density function whose continuous random variables are the pursuers' positions in W.

The only functionality that we require an abstract sampler to have is the ability to generate a point in W^n. The benefit of using an abstract sampler is that our algorithm is not dependent on a specific sampler to generate a solution strategy. This allows us to choose samplers that efficiently explore W^n. Note that the goal of catching the evaders means that the best sampling strategies may differ from those used in traditional motion planning algorithms. However, for our algorithm to be probabilistically complete, the abstract sampler must have a support equal to W^n (Section 7.2.4).

We demonstrate the feasibility of using an abstract sample generator in our algorithm by providing simulation results that utilize various sample generators (Section 7.3).

7.2.2 Constraints

In this section we discuss the constraints used in our main algorithm that determine whether a connection should be made between two vertices (Algorithm 13 line 7). The three constraints can be categorized as visibility, connection distance, and cycle distance constraints. The visibility constraint is required for correctness, whereas the connection and cycle distance constraints aim to reduce the time it takes for the algorithm to produce a feasible joint motion strategy.

Visibility Condition

The visibility condition states that for two vertices to share a pair of directed edges, the vertices corresponding joint pursuer configurations must be mutually visible to one another. This corresponds to the i^{th} pursuer of one configuration residing within the visibility region of the i^{th} pursuer in a neighboring configuration. Another way

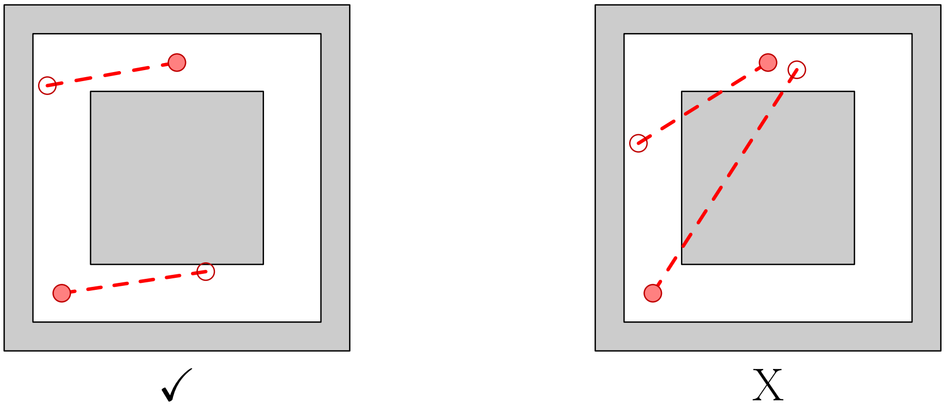


Figure 7.4: By considering only straight line motions that do not intersect the environment we ensure the generation of collision free strategies.

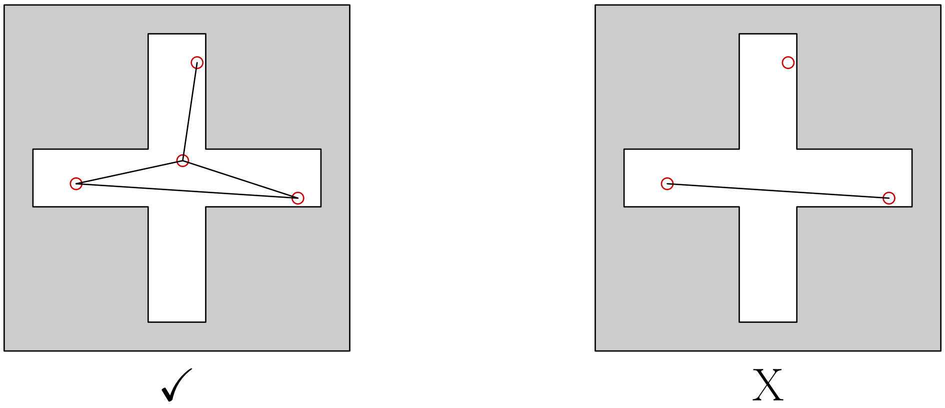


Figure 7.5: Multiple intermediary vertices are preferred to a single long connection.

of interpreting this constraint is that only straight line motions are permitted between corresponding pursuers in neighboring vertices. This constraint prevents the generation of strategies in which the pursuers collide with obstacles.

Edge Length

To limit the amount of time spent computing the reachable data when an edge is added in the SG-PEG we place a constraint on the length of the segment connecting the vertices joint pursuer configurations in W^n. The idea is that given two joint configurations that are far apart, requiring multiple intermediary vertices as opposed

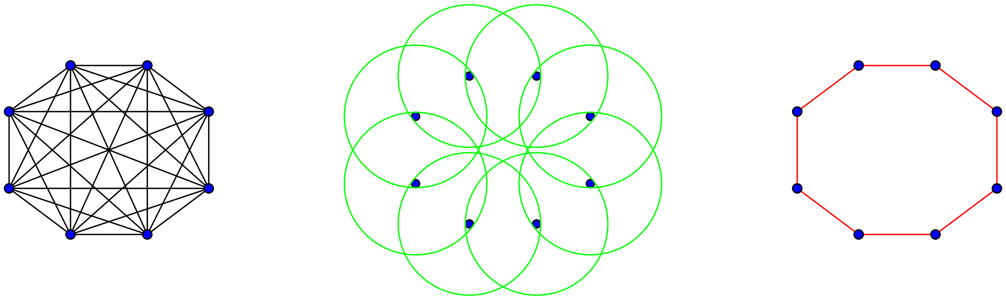


Figure 7.6: A collection of dense samples is problematic since cycles can occupy a large amount of computing resources. To combat this we can require all cycles to be a minimum length.

to a single long connection is preferred. The intermediary vertices provide additional opportunities for any potential subsequent samples to become connected.

Minimum Cycle Length

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

7.2.3 Search for a Solution Strategy

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

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

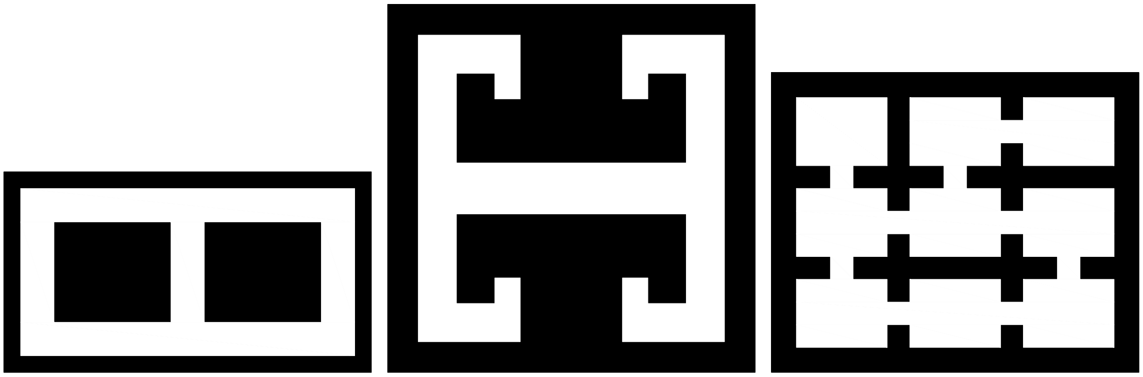
7.2.4 Probabilistic Completeness

Finally, we argue that under certain conditions Algorithm 13 is probabilistically complete.

Theorem 2. *If the abstract sampler has a support equal to W^n, and there are no constraints on the edge length and cycle length, then our algorithm is probabilistically complete. That is, the probability of our algorithm finding a solution, if one exists, tends to* 1 *as the number of samples goes to infinity.*

Proof. The argument proceeds in the same fashion as the probabilistic completeness proof for PRM presented by Kavraki, Kolountzakis, and Latombe [38]. The only significant difference is that, instead of considering the clearance between a solution strategy and the obstacle boundaries, we must consider the clearance from the critical boundaries at which shadow events that are not part of the final solution strategy would occur. □

Note that for our algorithm to maintain its probabilistic completeness, it may be necessary to relax the minimum cycle length constraint when deciding whether to insert an edge.



(a) Brick environment. (b) “H” environment. (c) Office room environment.

Figure 7.7: Environments used in our simulations.

7.3 Simulation Results

We implemented our algorithm in simulation and provide some results for three different environments, using three different sample generators, and three different cycle constraints. The environments (Figure 7.7) all require at least two pursuers to generate a solution strategy. As such we have deployed two pursuers to test our algorithm. The three different sample generators have the following behavior:

- SG_1 - Returns a uniform sample in W^n. This is a baseline sample generator that produces independent and identically distributed samples in F^n. This sample generator satisfies the completeness constraint.

- SG_2 - Selects a uniform random point in W for p_1. Each successive pursuer j is assigned a uniform random point in $W - \bigcup_{i<j} V(p_i)$, such that no two pursuers are mutually visible. If $W - \bigcup_{i<j} V(p_i) = \emptyset$ then the entire environment is viewable and any subsequent pursuers are assigned a random point in W. By ensuring that the pursuers cannot see one another, we maximize exploration by generating samples where the pursuers’ visibility regions don’t overlap. Note that this sample generator does not satisfy the completeness constraint.

- SG_3 - Selects an existing SG-PEG vertex, and for each pursuer selects a new target position from the pursuer's current visibility region. This is a local randomized sampler. By sampling within an existing SG-PEG vertex's field-of-view, we are essentially causing the search to "bloom" from the root vertex. This sample generator satisfies the completeness constraint.

For each combination of environment, sample generator, and cycle constraints we ran 10 trials, each with a unique starting position. Each simulation was given a maximum computation time limit of 1200 seconds. If the algorithm could not generate a solution strategy within the allotted time, we assumed that it failed.

The cycle constraints represent the extremes and one intermediary constraint. By not allowing any cycles, the SG-PEG has a tree structure, and may encounter environments where this limitation prevents our algorithm from generating a solution strategy until a sufficiently dense collection of samples are added to the SG-PEG. The other extreme has no constraint on potential cycles. This means that if the samples are close together, then our algorithm will spend a lot of time computing reachable shadow labels as opposed to exploring. However, this is a necessary condition for probabilistic completeness.

We report a number of statistics (Tables 7.1, 7.2, and 7.3) for each scenario. The first item that we report is the number of successes (was a solution strategy found) across all trials. For the following we report both the mean and standard deviation: the computation time in seconds, the number of SG-PEG vertices created, the number of reachable labels computed, and the total distance travelled by the pursuers.

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

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

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

There are two main conclusions that we can draw from our simulations. The first is the effect the cycle length constraint has on all of the metrics that appear in Tables 7.1, 7.2, and 7.3. When cycles were not allowed, the algorithm was able to generate a solution faster, requiring the pursuers to travel a shorter distance, often with a negligible increase in the number of vertices and reachable labels. When no constraints were placed on the cycle length, there was a noticeable decrease in performance. None of the samplers were able to generate a solution to the “office room” environment within the allotted time without the cycle length constraint.

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

Table 7.1: Simulation Results for the brick environment.

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

Table 7.2: Simulation Results for the H environment.

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

Table 7.3: Simulation Results for the room environment.

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

Chapter 8

Pursuit Evasion for a Single Pursuer with Fixed Beams

This chapter considers the problem of planning motions for a mobile robot equipped with a finite collection of single-direction sensors, with the goal of locating an adversarial evader within the line-of-sight of one of those sensors. This problem can viewed as a restricted version of several others in the literature, including the Guibas, Latombe, LaValle, Lin, and Motwani algorithm found in Chapter 4 (in which the robot has an omnidirectional sensor); Gerkey, Thrun, and Gordon [22] (in which the robot has an angle-bounded but continuous and rotatable field of view); and Kameda, Yamashita, and Suzuki [34] (in which the robot, called a 1-searcher, has a single rotatable beam sensor).

The unique restriction that we consider here is that the directions of the sensor, expressed in world coordinates, are *fixed*. The pursuer robot cannot rotate to aim its sensors in its search for the evader; it must locate the evader using only translations. The new contribution of this paper is a complete and efficient algorithm for solving this fixed-beam pursuit-evasion problem. We also present an implementation of this algorithm, and show computed examples demonstrating its correctness and effectiveness.

Figure 8.1 shows an example pursuit plan generated by our algorithm. In this instance, the pursuer has four beams, oriented in up, down, left, and right positions. (Our algorithm works for arbitrary collections of beams, not just orthogonal ones.)

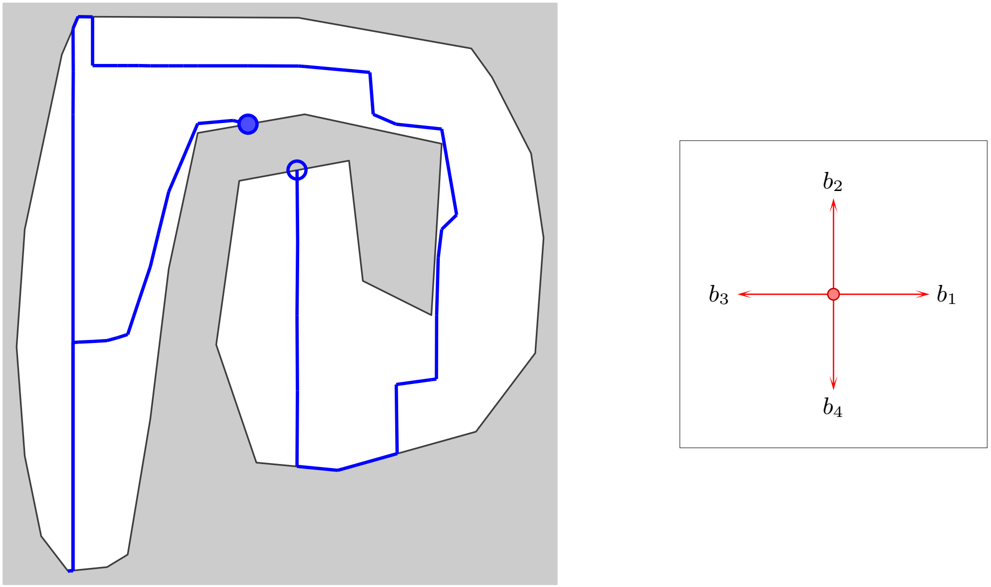


Figure 8.1: A pursuit plan (left) computed by our algorithm. The pursuer uses four orthogonal beams (right) to capture the evader, regardless of the evader's path or velocity. The pursuer starts on the bottom boundary of the top-center corridor, and travels first to the left and then to the right.

Starting from the bottom of the middle section of the spiral, the pursuer travels left to the end, then back around to the center of the spiral. In several cases, the robot moves to the boundary of the environment, which is necessary in this problem to ensure that an evader is captured. If the area between two beams has non-zero area, then it is possible for the evader to hide there indefinitely. Our algorithm is complete, in the sense that if a path exists to clear the given environment with the given beams, we are certain to find it. If no such path exists, the algorithm terminates with a failure result.

Our work on this problem is motivated by several related factors. First, and most importantly, it provides another data point for understanding the computational, sensing, and movement requirements that underlie the problem of searching for evaders. There is an obvious connection between the pursuer's sensing and movement capabilities and the existence of a solution. Any instance that can be solved with weak sensors can also be solved with relatively stronger sensors [61]. The relationship between sensing capabilities and the computation needed for planning is less obvious. For example, compared to the existing algorithm for omnidirectional sensing [27], the computation time is *increased* (due to an additional dimension of the underlying C-space) by a restriction to a rotatable range of sensing angle [22], but *decreased* (down to constant memory and constant time to compute the next movement) by a further restriction of that range to a single rotatable direction. Our results, which include an algorithm whose run time is polynomial in the complexity of the environment but exponential in the number of sensors, suggest that the computational difficulty of these kinds of problems is governed not just by the informative value of the sensors, but also by the *complexity* of that sensor model, measured informally by the non-trivial relationships between information that is observable and information that is unobserved.

We also suspect that there may be some direct value to studying restricted versions of planning problems, as an algorithmic tool for solving the original, unrestricted problems. The idea draws inspiration from the Miller-Rabin primality-testing algorithm [52, 72], which employs a probabilistic test that determines, with a known success probability, whether a given number is ‘definitely composite’ or ‘possibly prime’. The algorithm works by iterating this test, until the input integer is demonstrated to be composite, or until its probability of being composite in spite of repeatedly passing the test becomes acceptably small. We are likewise interested in planning algorithms that attack challenging problems by attempting to solve a randomly-generated series of restricted, but efficiently-solvable, related instances. Under the right conditions, we can ensure that if any of those restricted instances has a solution, the solution applies to the original problem as well. The algorithm proposed here is a first step toward applying that strategy to the full omnidirectional visibility-based pursuit evasion problem.

The remainder of this chapter is laid out as follows. Section 8.1 revisits the problem formulation initially introduced in Chapter 2 to address the changes to the sensor footprint due to the robot's fixed beam sensors. The algorithm details appear in Section 8.2. Section 8.3 describes our implementation and presents some computed examples. Concluding remarks appear in Section 8.4.

A preliminary version of this work is to appear in [87].

8.1 Problem Formulation: Fixed Beams

As opposed to revisiting the entire problem formulation from Chapter 2, this section will focus specifically on the fixed-beam sensor model and will reframe the capture condition to account for the change in sensor model.

The pursuer is equipped with a set B of m **beam sensors**, each of which can detect the evader by line-of-sight in a single, fixed direction. We represent these directions as a collection of unit vectors $B = \{b_1, \ldots, b_m\}$. These directions remain constant as the pursuer moves; the pursuer cannot rotate them. A beam sensor $b_i \in B$ **detects** the evader at time t if there exists a non-negative scalar a such that $e(t) = p(t) + ab_i$ and the line segment connecting $p(t)$ and $e(t)$ is fully contained in W, that is, if $\overline{p(t)e(t)} \subset W$.

The inputs to our algorithm are an environment W, a set of beams $B = \{b_1, \ldots, b_m\}$, and a starting pursuer position $p(0) \in W$. The goal is to compute a continuous finite-length path $p : [0, T] \rightarrow W$ for the pursuer that guarantees that at least one beam will detect the evader, or report that no such path exists. That is, the algorithm should generate a path p starting from the given $p(0)$, and a termination time T, such that for any continuous evader path e, there exists some time $t \leq T$ and some beam $b_i \in B$ such that beam b_i detects the evader at time t.

8.2 Description of Algorithm

This section describes an algorithm for the pursuit-evasion problem introduced in Section 8.1. Although, at a very high level, the structure of the algorithm follows the same form as existing algorithms for related problems [22, 27, 85], this algorithm differs substantially in its important details. The intuition is to keep track, using a small collection of boolean labels, of which portions of the environment might contain the evader if it has not yet been detected. We partition the environment into a finite set of regions called conservative regions, within each of which the labels remain constant, and track how the labels change as the robot moves between those conservative regions. This induces a graph, through which the algorithm searches for a path from the node representing the initial condition to one in which the evader has certainly been captured. The remainder of this section describes the details.

8.2.1 Gaps

In general, the pursuer's m beam sensors divide the environment into a collection of m regions, called **gaps**, that are not currently detectable by any of those beams. More precisely, a **gap** is a maximal path-connected component of the environment that does not cross any of the beams. A **gap** in this context is synonymous to the concept of a **shadow** that was introduced in Section 2.2 for pursuer(s) with omnidirectional sensing capabilities.

Note that, if the pursuer is in the interior of W, then each gap includes two beams on its boundary. We write g_{ij} to denote the gap whose boundary includes b_i and b_j. Figure 8.2 illustrates the above notation.

The important idea is that the evader, if it has not been captured, is always contained in exactly one gap, in which it can move freely. Although the pursuer does not know the evader's position, it can infer, based on its prior movements, whether

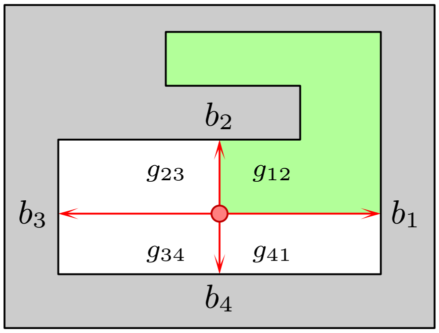


Figure 8.2: A robot with four fixed beam sensors. In this example gap g_{12} is shaded green.

an evader could potentially reside within each gap.

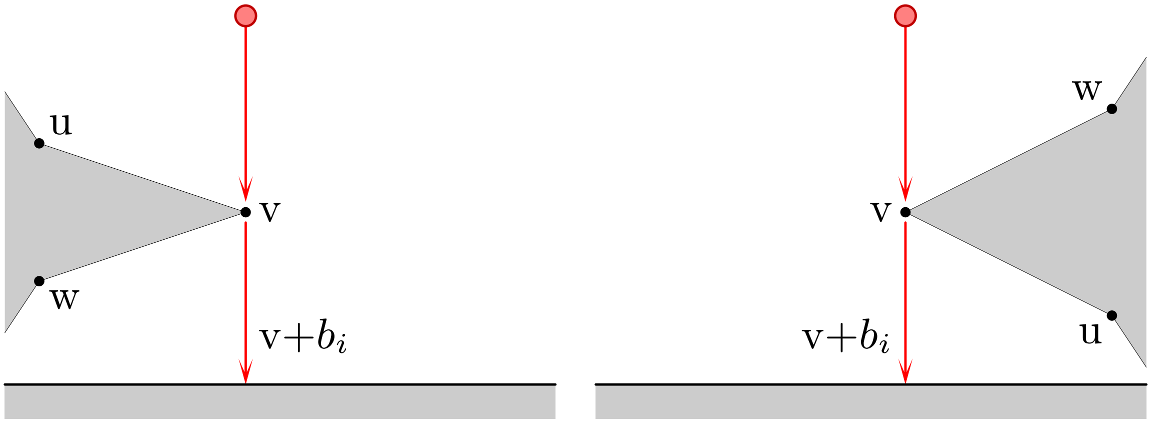
A gap g_{ij} is **cleared** at time t if, based on the pursuers' motions up to time t, it is not possible for the evader to be within g_{ij} without having been captured. A gap is **contaminated** if it is not clear. That is, a contaminated gap is one in which the evader may possibly reside. We assign a binary label to each gap corresponding to its cleared/contaminated status. A label of 0 means that the gap is cleared; a label of 1 means that the gap is contaminated.

Notice that, since the evader can move arbitrarily quickly, the pursuer cannot draw any more detailed conclusion about each gap other than its clear/contaminated status; if any part of a gap can contain the evader, then the entire gap is contaminated. As a result, we can encapsulate all of the information available to the pursuer by tracking only the pursuer's current configuration and the current gap labels.

8.2.2 Decomposition into Convex Conservative Regions

The algorithm begins by decomposing the environment into a collection of convex conservative regions. A region $R \subset W$ is **conservative** if the gap labels (clear or contaminated) remain unchanged as the pursuer moves within R.

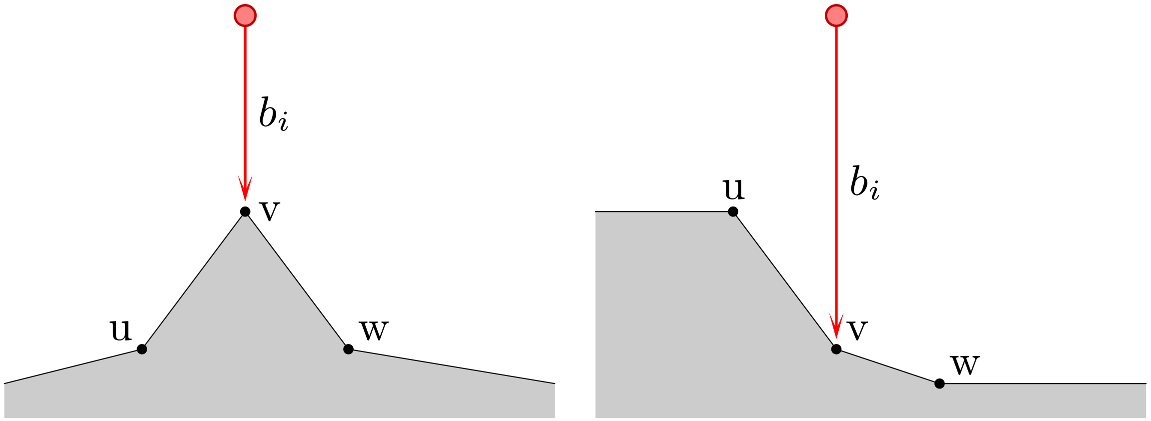
First, we partition the environment into conservative regions by identifying segments in the interior of W at which changes to the gap labels can occur. For a given



Left events

Right events

Figure 8.3: An illustration to detect when a **Left** event occurs. An illustration to detect when a **Right** event occurs.



No event (reflex)

No event (convex)

Figure 8.4: An illustration that demonstrates when an event does not occur at a reflex vertex. An illustration that demonstrates when an event does not occur at a convex vertex.

beam $b_i \in B$, the crucial locations for the pursuer are positions $p(t)$ at which the ray extension $p(t) + ab_i$ within W ends at a vertex v of W. At such points, the distance observed by the beam can change discontinuously, potentially allowing the evader to transit from one gap to another.

There are three distinct cases, of which only two can cause a change in the pursuer's gap labels (Figure 8.3), and one is safely ignored (Figure 8.4). We distinguish these cases via clockwise (cw) and counterclockwise (ccw) tests involving the vertex

v and its immediate predecessor u and successor w (in clockwise order) along ∂W.

1. A **left** event occurs when the following condition is satisfied:

   $$\mathrm{cw}(\mathrm{v}, \mathrm{v} + b_i, \mathrm{u}) \text{ and } \mathrm{cw}(\mathrm{v}, \mathrm{v} + b_i, \mathrm{w}).$$

   That is, left events are generated when the boundary curve at v is on the left side of $v + b_i$.

2. A **right** event occurs when the following condition is satisfied:

   $$\mathrm{ccw}(\mathrm{v}, \mathrm{v} + b_i, \mathrm{u}) \text{ and } \mathrm{ccw}(\mathrm{v}, \mathrm{v} + b_i, \mathrm{w}).$$

   Right events occur when the boundary curve at v is on the right side of $v + b_i$.

3. The case in which u and w are on opposite sides of a beam does not generate any change to the gap labels because b_i changes continuously at this point, regardless of whether v is a convex or reflex vertex.

For each vertex $v \in W$ and $b_i \in B$, the set of pursuer positions that generate such events can be found by extending a ray within W, starting at v, in direction $-b_i$.

If we additionally know the direction (that is, forward or backward) with which the pursuer crosses this critical event, we can classify the event further.

- If the pursuer is moving so that the endpoint of b_i sweeps across $\overline{uv}$ before reaching v, then after crossing v, the beam will **extend** (or “embiggen”) to a new environment edge.

- Conversely, if b_i crosses v in the opposite direction, the beam will **retract** (or “unembiggen”) to environment edge $\overline{uv}$.

By performing these ray extensions, the algorithm forms a decomposition of the environment into conservative regions. See Figure 8.5. We represent this decomposition as a doubly-connected edge list (DCEL). Each interior half-edge in the DCEL is

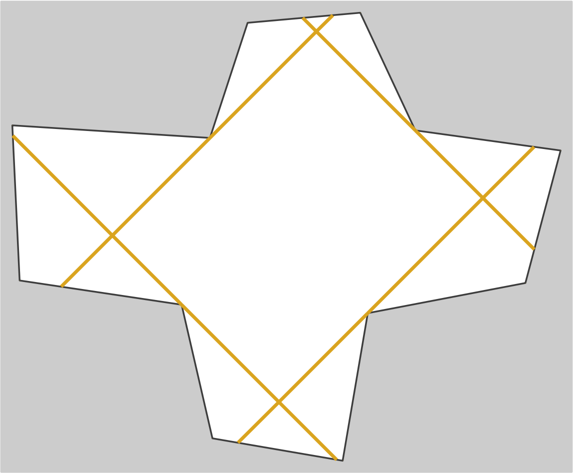


Figure 8.5: Decomposition of a simple environment into conservative regions by ray extensions.

labeled with the event type (left or right; extend or retract) and the generating beam b_i.

Note in particular that, although this decomposition generates regions that are conservative, those conservative regions are not necessarily convex. To enable straightforward generation of a path from a sequence of adjacent regions (which will be the final step of the algorithm; see below), we refine the partition via trapezoidal decomposition, ensuring that every region is convex. Figure 8.6 shows an example of the final decomposition. Half-edges added at this stage are not labeled with any events.

8.2.3 Fixed-Beam Pursuit-Evasion Graph

In this section, we describe our algorithm which utilizes the convex conservative decomposition to construct its primary data structure, called the Fixed-Beam Pursuit-Evasion Graph (FB-PEG). This section describes the vertices and edges the FB-PEG.

The basic idea is that each node of the FB-PEG corresponds to one element of the convex conservative decomposition including interior faces and the edges and vertices that surround each face, with a few important exceptions.

- The unbounded face of the DCEL, which represents the obstacle region $\mathbb{R}^2 - W$,

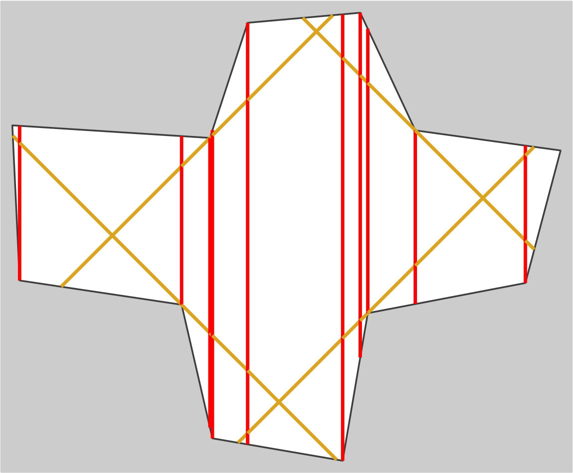


Figure 8.6: Refinement of the decomposition from Figure 8.5 by vertical ray extensions upward and downward from each vertex. The resulting cells are convex.

does not generate any FB-PEG nodes.

- The halfedges and vertices created during the decomposition do not generate FB-PEG nodes, because many of these correspond to positions at which the gaps are changing. For clarity, we instead include FB-PEG edges that transition directly between the associated faces (resp. edges) without stopping at the dividing half-edge (resp. vertex).

All other elements of the DCEL generate FB-PEG nodes. This detail is important, because without coming into contact with the boundary of W, the pursuer can never clear any gaps.

Along with a specific convex conservative region, each FB-PEG node is also associated with a unique set of clear/contaminated labels for each of the gaps that exist in that region. (Note that, because these regions are conservative, the set of gaps is the same for every point within a given region.) Thus, each bounded face in the DCEL generates 2^m nodes in the FB-PEG; depending on the directions of the beams and the environment boundary, there will be between 2 and 2^{m+1} FB-PEG nodes associated with each edge and vertex represented in the graph.

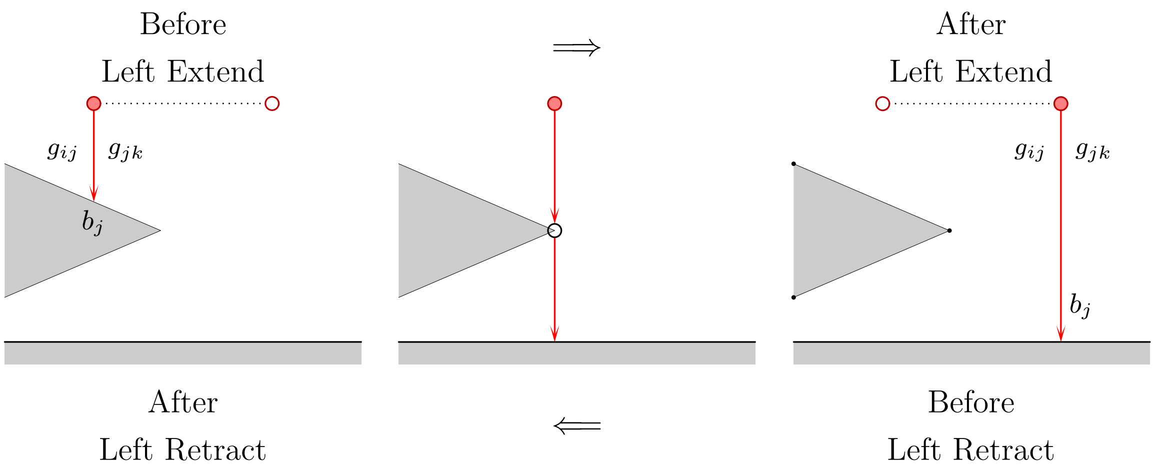


Figure 8.7: An illustration of the **Left** Extend/Retract events.

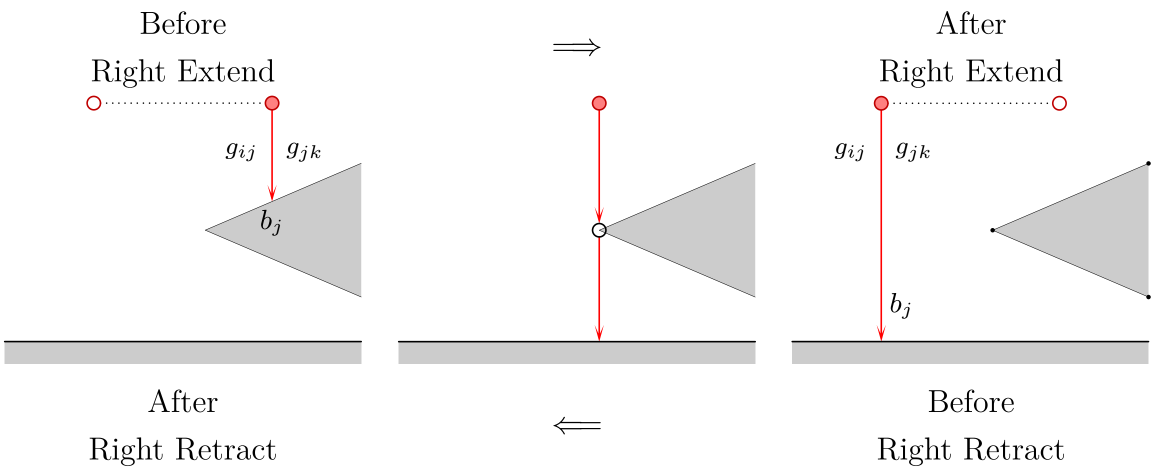


Figure 8.8: An illustration of the **Right** Extend/Retract events.

To compute the edges of the FB-PEG, we iterate over each FB-PEG node (which is already associated with gap labels) and consider each of its neighbors in the DCEL. It remains only to determine which FB-PEG node for that region the directed edge should connect to. We can categorize the gap update rules that occur when transitioning between a source FB-PEG node and a target FB-PEG node into one of five cases, based on the dimension—that is, face (F), edge (E), or vertex (V)—of the source and target nodes.

$F \to F$ and $E \to E$ transitions

When the pursuer transitions from a face to an adjacent face (skipping a ray extension in the interior of W) or from an edge to an adjacent edge (skipping the endpoint of such a ray extension), we must update the clear/contaminated labels for the appropriate gaps. These transitions occur at the Left and Right events described in Section 8.2.2. There are four different event types leading to two different kinds of update rules.

1. At a Left-Extend event, the evader can hide behind the obstacle touched by b_i until after the beam has passed, and then contaminate the gap to the left of b_i. At a Right-Retract event the same effect occurs in reverse. Thus when the pursuer passes a Left-Extend or Right-Retract event from beam b_i, we assign:

   $$g_{i-1} \leftarrow (g_{i-1} \ \text{ or } \ g_i).$$

   All other gaps retain the same labels.

2. For Left-Retract and Right-Extend events, the cross contamination occurs in the opposite direction, so instead we assign:

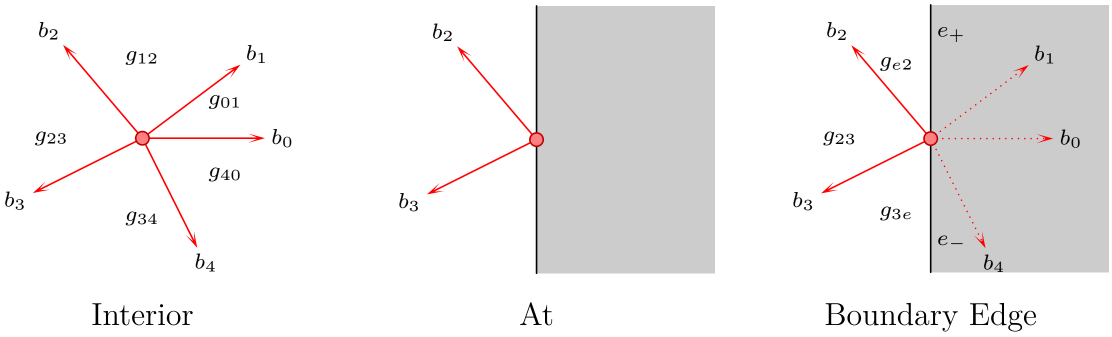
   $$g_i \leftarrow (g_i \ \text{ or } \ g_{i-1}).$$

   Again, the other gaps do not change at this transition, so their gap labels remain unchanged.

Figures 8.7 and 8.8 illustrate these update rules.

$F \to E$ and $E \to F$ transitions

A second class of transitions moves from the interior of W to a boundary edge, or back again. Such movements can change the set of gaps in a variety of ways:



Interior At Boundary Edge

Figure 8.9: A scenario where gap edges can appear/disappear and shrink/grow. When travelling from the interior to the boundary edge g_{01} and g_{40} disappear, g_{12} and g_{34} shrink to g_{e2} and g_{3e}, and g_{23} remains the same. Conversely, when travelling from the boundary edge to the interior gap edges appear, grow, and remain the same.

- A gap can appear or disappear (when both of its incident beams are aimed directly into the wall).
- A gap can shrink or grow (when only part of the gap is pressed against the wall).
- A gap can split into multiple gaps (when a wide gap is separated into two parts by coming into contact with the environment boundary).
- Multiple gaps can merge into a single gap (when a gap is re-joined with itself after leaving the environment boundary).

To handle all of these cases in a clean and compact way, we use a series of **gap containment** tests that determine whether the interior of one gap overlaps the interior of another. Such tests can be performed in constant time using a series of cw and ccw tests on the beam vectors. Then each gap in the target FB-PEG node is marked as contaminated if and only if it overlaps at least on contaminated gap for the source node. Figure 8.9 illustrates one of these transitions.

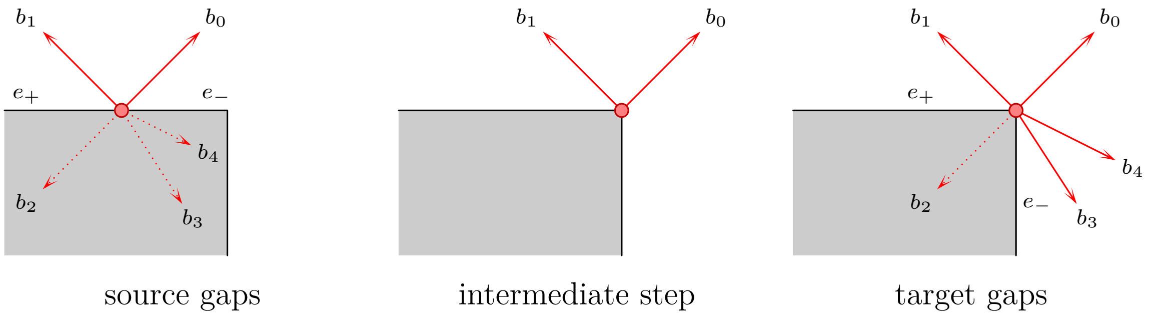


Figure 8.10: An illustration of the **split** and **merge** events that occur when transitioning between an edge of the region graph and an environment vertex.

$V \to E$ and $E \to V$ transitions

Transitions from an edge to a vertex or from a vertex to an edge can be handled identically to the $F \to E$ and $E \to F$ cases, with one important exception: If the vertex is a reflex vertex, then it is possible that some beams will emerge from (or disappear into) the environment instantaneously (rather than the gradual appear/disappear changes that occur for F/E transitions).

To handle this case properly, we must introduce an intermediate step, in which the gaps are computed at the reflex vertex, but only for beams which do not extend into the environment boundary in *both* the source and target regions. Figure 8.10 illustrates one of these transitions, for which the intermediate step is computed at the environment vertex. We use a series of gap overlap tests to propagate contamination forward from the source FB-PEG node, correctly allowing the move past beams that are blocked by the edge (whether it is the source or target node). We then use the clear/contaminated labels from these intermediate gaps to populate the labels in the target node.

$F \to V$ and $V \to F$ transitions

Our algorithm omits direct transitions between faces and vertices for simplicity. Particularly for the case of reflex vertices, the correct assignment of gap labels is non-trivial, because the algorithm must correctly identify which events to apply. This omission does not impact the correctness nor completeness of the algorithm, because any solution that traverses directly between a vertex and face can achieve the same result indirectly via the corresponding edge. (This does, of course, potentially make some of the final paths slightly longer.)

$V \to V$ transitions

The final of the nine cases is $V \to V$, which cannot occur because vertices are never adjacent in a DCEL.

Taken together, this set of nodes and edges fully captures the possibilities for the evader's location as a function of the pursuer's movements across the conservative regions.

8.2.4 Path Generation

The final step of the algorithm is a forward search through the FB-PEG. The search starts from the pursuer's initial position with all of the gaps labeled as contaminated, and terminates when it reaches a FB-PEG node in which all of the gaps are labeled clear. We use breadth-first search, though any graph search would be suitable.

Given a path from all-contaminated to all-clear, we generate the pursuer's final path in the usual way for cell-decomposition-based planning: We chain together the centroids of each region visited along the FB-PEG path. The only complication is that, for $F \to F$ transitions, we must also include the midpoint of the edge separating those faces. The resulting path is guaranteed to locate the evader. Since the regions are all convex, the resulting path is guaranteed to stay within W, and to visit the FB-PEG nodes in the correct order. Figure 8.11 completes the running example from Figures 8.5 and 8.6, illustrating a plan that correctly locates the evader.

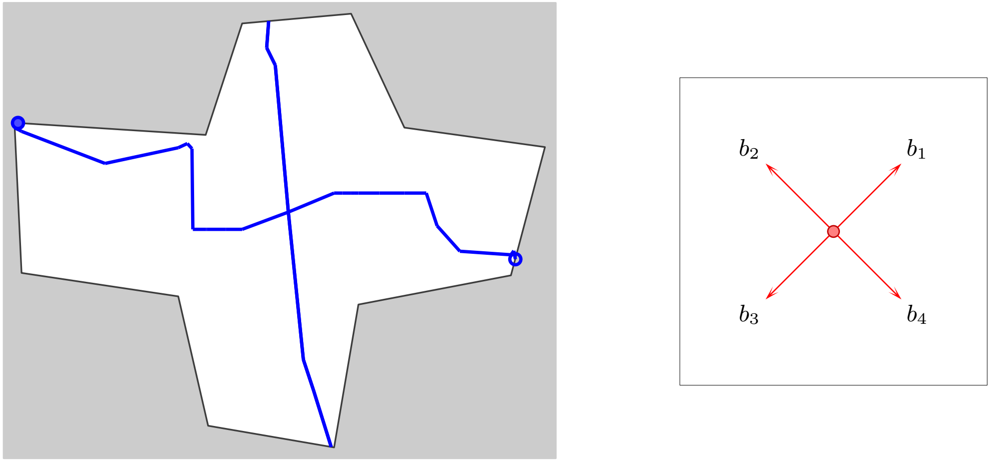


Figure 8.11: The final generated plan for the example shown in Figures 8.5 and 8.6.

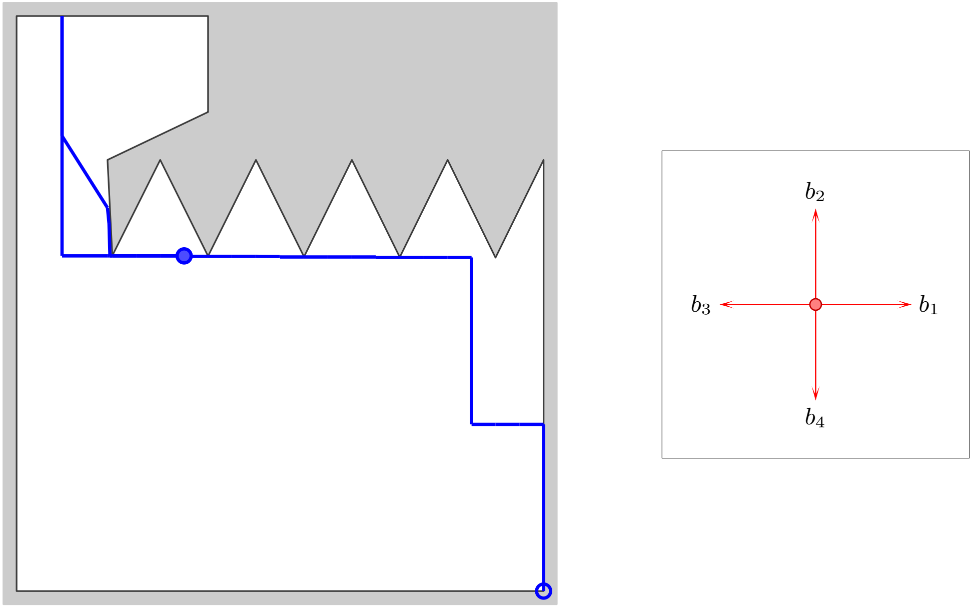


Figure 8.12: A plan generated by our algorithm for the above environment.

Finally, because we know that the sequence of conservative regions is sufficient to characterize a solution, we know that if the FB-PEG does not contain a path from the start node to an all-clear node, then the underlying pursuit-evasion problem has no solution.

8.2.5 Runtime analysis

The run time of this algorithm is dominated by the time needed to search the FB-PEG, which has $O(2^m n^2)$ nodes and $O(2^m n^2)$ total edges. Therefore, the algorithm takes time $O(|V| + |E|) = O(2^m n^2)$. Note that this runtime is polynomial in the complexity of the environment.

8.3 Simulation Results

This section presents some example pursuit strategies computed by our implementation of this algorithm. Three examples appear in Figure 8.1, 8.11, and 8.12. With this implementation, which uses C++, a machine utilizing a single core of an Intel i5 processor and running the Gnu/Linux operating system was able to solve each of these instances in less than 0.1 seconds.

8.4 Conclusion

In this chapter we present a complete algorithm for solving a pursuit-evasion problem in a simply-connected two-dimensional environment, for the case of a single pursuer equipped with fixed beam sensors. The algorithm constructs a DCEL by decomposing the environment based on critical gap events and further refines the partition by employing a trapezoidal decomposition to ensure a convex conservative decomposition. The decomposition induces a FB-PEG, which is exhaustively searched and returns either a path through the FB-PEG which corresponds to a pursuer motion strategy through the environment which is guaranteed to capture an evader, or reports failure for the current beam configuration.

Chapter 9

Discussion and Conclusion

This thesis began by contending that the rate at which robots and other autonomous agents are adopted into various application domains hinges upon the availability of robust and efficient algorithms that are capable of solving the complex planning problems inherent to the given domain. We have shown that there are a wide array of tasks that can be framed as pursuit-evasion problems such as surveillance [8,56],search and rescue [24, 83], and missile-guidance systems [36, 71]. This thesis provides several theoretical results which can hopefully be used to expedite the procurement of robust planners and algorithms for use on physical systems that would enable them to operate in application domains that have previously had little use for autonomous systems due to the task complexity.

The remainder of this chapter contains some discussion which is meant to place the results of this thesis into context as well as some interesting open problems. Section 9.1 revisits the family of visibility-based pursuit-evasion problems found in this thesis and highlights the contributions, limitations, and open problems that remain to be solved. We conclude in Section 9.2 with a discussion of possible future directions.

9.1 Contributions, Limitations, and Open Questions

In this section we put the results of this thesis into perspective by taking a chronological look at the novel contributions, limitations, and remaining open questions for each of the visibility-based pursuit-evasion problems found in the text.

The three unique visibility-based pursuit-evasion problems for which we provide novel results are:

- A Single Pursuer with an omni-directional sensor that extends to the polygonal boundary (Chapter 5).
- Multiple Pursuers with an omni-directional sensor that extends to the polygonal boundary (Chapters 6 and 7).
- A Single Pursuer whose sensors are comprised of a finite collection of fixed beams (Chapter 8).

Single Pursuer - Optimal 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 cost solution strategy. Ample effort has already been put towards identifying necessary and sufficient pruning of suboptimal paths when conducting the forward search. Due to the exponential nature of the graph, 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 [94, 95] 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.

Multiple Pursuers This thesis provided two novel results for the multi-pursuer visibility-based pursuit-evasion problem. The first result was a complete algorithm (Chapter 6) that identified the critical visibility events that can occur when a group of pursuers move within a 2-dimensional polygonal environment. The major drawback of this work is that it suffers from the "curse of dimensionality" [7] since the complete algorithm is doubly-exponential. The second result in this line of research is able to overcome this problem by sampling from the pursuers' joint configuration space. The result is a sampling-based algorithm (Chapter 7) that is capable of generating a joint motion strategy for the pursuers that captures an evader.

The most immediate candidate for improving the existing algorithm is to investigate how the algorithm updates shadow labels when travelling from one joint pursuer configuration to another. Currently, the algorithm updates the shadow labels numerically (Algorithm 2). An alternative approach is to solve for the critical events analytically. The initial and goal configurations are known so it should be possible to parameterize the equations, with time as the parameter, and solve for the critical events analytically.

The more far-reaching problem is the development of robust sampling strategies for the visibility-based pursuit-evasion[1] domain, which poses more difficulty than other domains because it is a complex configuration space that has restrictive constraints (capture guarantees). Typically, the ability to both draw a random sample in

[1] To the author's knowledge there hasn't been any work specifically focused on sampling in the visibility-based pursuit-evasion domain. There has been nominal research done that utilizes sampling techniques to consider the differential game variant of the pursuit-evasion problem [35].

the configuration space and perform any connections to the underlying data structure are both relatively cheap operations. However, in the visibility-based pursuit-evasion problem, the “connection” phase takes substantially longer than the sampling phase due to the additional complexity which motivates the question: “What makes one configuration more preferable to another?” Any insight gained by answering this question could be useful when considering other planning problems that currently can not be solved using a straightforward application of traditional sampling-based planners.

Single Pursuer - Fixed Beam The third problem for which we have a result is the fixed-beam visibility-based pursuit-evasion problem. Recall, the fixed-beam variation of the pursuit-evasion problem is a restricted formulation where the directions of the pursuers’ sensors, expressed in world coordinates, are fixed. The pursuer robot cannot rotate to aim its sensors in its search for the evader; it must locate the evader using only translations. We designed an algorithm that is capable of solving the fixed-beam pursuit-evasion problem.

It remains to show if there are any benefits to using restricted versions of planning problems as an algorithmic tool for solving the original unrestricted problems. The idea, which draws inspiration from the Miller-Rabin primality testing algorithm, seems like a promising avenue for those that study motion planning problems. Current algorithms seem ill-suited to this level of exploitation since the computational difficulty for the visibility-based pursuit-evasion problem has been shown to be not just reliant on the informative value of the sensors, but also on the complexity of the sensor model.

9.2 Future Directions

There are two major obstacles that are currently hindering the application of visibility-based pursuit-evasion algorithms on physical systems. The first is the strong sensing requirements that are required to carry out the search. The second is a lack of algorithms that are robust to sensor failure. These problems independently pose enough of a hurdle to make the visibility-based pursuit-evasion problem more difficult, but when considering a physical system both problems will need to be addressed.

Sensing Requirements As alluded to in the Introduction to Chapter 8, the Robotics community still does not have a firm grasp as to the sensing requirements that underlie the search task. Many current algorithms assume a sensing model where the pursuer has an omnidirectional field-of-view which extends to a polygonal boundary. This model may be practical in indoor environments where some combination of camera system and Lidar is able to replicate this model. However, even the most state of the art Lidar sensors will have trouble aspiring to this sensor model in outdoor environments because of the sensors range limitations.

Sensor Failures A common assumption in visibility-based pursuit-evasion is that the sensors used to detect evaders are perfectly reliable. The sensor model assumes that if the evader is within view of any pursuer for any positive time interval then it will be detected during that entire interval. This assumption is extremely problematic because, when implemented on real sensors systems, such plans cannot account for the possibility of short-term false negative errors in evader detection. Depending on the evader model that the algorithm employs, the introduction of even the smallest of sensing errors can render the algorithm useless (the case when evaders have finite unbounded speed).

Bibliography

[1] B. Alspach. Searching and sweeping graphs: a brief survey. *Matematiche*, 59:5–37, 2004.

[2] D. S. Arnon. A cellular decomposition algorithm for semi-algebraic sets. In *EUROSAM*, volume 72 of *Lecture Notes in Computer Science*, pages 301–315. Springer, 1979.

[3] J. Aslam, Z. Butler, F. Constantin, V. Crespi, G. Cybenko, and D. Rus. Tracking a moving object with a binary sensor network. In *Proc. International Conference on Embedded Networked Sensor Systems*, pages 150–161, New York, NY, USA, 2003. ACM.

[4] T. Bandyopadhyay, Y. P. Li, M. H. Ang Jr., and D. Hsu. Stealth tracking of an unpredictable target among obstacles. In *Proc. Workshop on the Algorithmic Foundations of Robotics*, pages 43–58, 2004.

[5] M. Batalin and G. S. Sukhatme. Sensor coverage using mobile robots and stationary nodes. In *SPIE Conference on Scalability and Traffic Control in IP Networks II (Disaster Recovery Networks)*, volume 4868, pages 269–276, August 2002.

[6] M. Batalin and G. S. Sukhatme. The analysis of an efficient algorithm for robot coverage and exploration based on sensor network deployment. In *Proc. IEEE International Conference on Robotics and Automation*, pages 3489–3496, April 2005.

[7] R. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[8] S. Bhattacharya, T. Başar, and M. Falcone. Surveillance for security as a pursuit-evasion game. In R. Poovendran and W. Saad, editors, *Decision and Game Theory for Security*, pages 370–379. Springer International Publishing, 2014.

[9] D. Bienstock. Graph searching, path-width, tree-width and related problems (A Survey). In F. Roberts, F. Hwang, and C. Monma, editors, *Reliability Of Com-*

puter And Communication Networks, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, December 2-4, 1989, volume 5 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. DIMACS/AMS, 1991.

[10] R. Borie, S. Koenig, and C. Tovey. Pursuit-evasion problems. In J. Gross, J. Yellen, and P. Zhang, editors, *Handbook of Graph Theory*, chapter 9.5, pages 1145–1165. Chapman and Hall, 2013.

[11] J. Canny and J. Reif. New lower bound techniques for robot motion planning problems. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 49–60, 1987.

[12] S. Carlsson, H. Jonsson, and J. B. Nilsson. Finding the shortest watchman route in a simple polygon. *Discrete and Computational Geometry*, 22(3):377–402, 1999.

[13] G. Casella and R. Berger. *Statistical Inference*. Duxbury Resource Center, 2001.

[14] P. Chen, S. Oh, M. Manzo, B. Sinopoli, C. Sharp, K. Whitehouse, O. Tolle, J. Jeong, P. Dutta, J. Hui, S. Schaffert, K. Sukun, J. Taneja, B. Zhu, T. Roosta, M. Howard, D. Culler, and S. Sastry. Instrumenting wireless sensor networks for real-time surveillance. In *Proc. IEEE International Conference on Robotics and Automation*, pages 3128–3133, 2006.

[15] W. Chin and S. Ntafos. Optimum watchman routes. In *Proc. ACM Symposium on Computational Geometry*, pages 24–33, New York, NY, USA, 1986. ACM.

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

[17] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, Boston, MA, 2005.

[18] G. E. Collins. Hauptvortrag: Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In H. Barkhage, editor, *Automata Theory and Formal Languages*, volume 33 of *Lecture Notes in Computer Science*, pages 134–183. Springer, 1975.

[19] M. Dror, A. Efrat, A. Lubiw, and J. S. B. Mitchell. Touring a sequence of polygons. In *Proc. ACM Symposium on Theory of Computing*, pages 473–482. ACM Press, 2003.

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

[21] F. V. Fomin and D. M. Thilikos. An annotated bibliography on guaranteed graph searching. *Theoretical Computer Science*, 399(3):236–245, June 2008.

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

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

[24] M. A. Goodrich, B. S. Morse, D. Gerhardt, J. L. Cooper, M. Quigley, J. A. Adams, and C. Humphrey. Supporting wilderness search and rescue using a camera-equipped mini uav: Research articles. *Journal of Field Robotics*, 25(1-2):89–110, January 2008.

[25] C. Gui and P. Mohapatra. Power conservation and quality of surveillance in target tracking sensor networks. In *Proc. International Conference on Mobile Computing and Networking*, pages 129–143, 2004.

[26] L. Guibas. Sensing, tracking, and reasoning with relations. *IEEE Signal Processing Magazine*, 19(2):72–85, 2002.

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

[28] T. He, S. Krishnamurthy, J. Stankovic, T. Abdelzaher, L. Luo, R. Stoleru, T. Yan, L. Gu, J. Hui, and B. Krogh. Energy-efficient surveillance system using wireless sensor networks. In *Proc. International Conference on Mobile systems, Applications, and Services*, pages 270–283, New York, NY, USA, 2004. ACM.

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

[30] D. Hsu, W.S. Lee, and N. Rong. A point-based POMDP planner for target tracking. In *Proc. IEEE International Conference on Robotics and Automation*, pages 2644–2650, 2008.

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

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

[33] B. Jung and G. S. Sukhatme. Cooperative multi-robot target tracking. In *Proceedings of the International Symposium on Distributed Autonomous Robotic Systems*, pages 81–90, July 2006.

[34] T. Kameda, M. Yamashita, and I. Suzuki. On-line polygon search by a seven-state boundary 1-searcher. *IEEE Transactions on Robotics*, 22(3):446–460, June 2006.

[35] S. Karaman and E. Frazzoli. Incremental sampling-based algorithms for a class of pursuit-evasion games. In *Proc. Workshop on the Algorithmic Foundations of Robotics*, December 2010.

[36] J. Karelahti, K. Virtanen, and T. Raivio. Near-optimal missile avoidance trajectories via receding horizon control. *Journal of Guidance, Control, and Dynamics*, 30(5):1287–1298, 2007.

[37] N. Karnad and V. Isler. Lion and man game in the presence of a circular obstacle. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009.

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

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

[40] W. Kim, K. Mechitov, J. Choi, and S. Ham. On target tracking with binary proximity sensors. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 40, 2005.

[41] K. Klein and S. Suri. Capture bounds for visibility-based pursuit evasion. In *Proc. ACM Symposium on Computational Geometry*, pages 329–338, 2013.

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

[43] A. Kolling and S. Carpin. Surveillance strategies for target detection with sweep lines. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5821–5827, 2009.

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

[45] X. Lan and M. Schwager. Planning periodic persistent monitoring trajectories for sensing robots in gaussian random fields. In *Proc. IEEE International Conference on Robotics and Automation*, pages 2407–2412, May 2013.

[46] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic, 1990.

[47] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[48] S. M. LaValle, H. H. González-Baños, C. Becker, and J.-C. Latombe. Motion strategies for maintaining visibility of a moving target. In *Proc. IEEE International Conference on Robotics and Automation*, pages 731–736, 1997.

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

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

[51] N. Michael, E. Stump, and K. Mohta. Persistent surveillance with a team of mavs. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2708–2714, 2011.

[52] G. L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976.

[53] J. S. B. Mitchell. Approximating watchman routes. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 844–855, 2013.

[54] R. Murrieta, A. Sarmiento, S. Bhattacharya, and S. A. Hutchinson. Maintaining visibility of a moving target at a fixed distance: The case of observer bounded speed. In *Proc. IEEE International Conference on Robotics and Automation*, 2004.

[55] R. Murrieta-Cid, H. H. Gonzalez-Banos, and B. Tovar. A reactive motion planner to maintain visibility of unpredictable targets. In *Proc. IEEE International Conference on Robotics and Automation*, 2002.

[56] R. Murrieta-Cid, T. Muppirala, A. Sarmiento, S. Bhattacharya, and S. Hutchinson. Surveillance strategies for a pursuer with finite sensor range. *International Journal of Robotics Research*, 26(3):233–253, 2007.

[57] R. Murrieta-Cid, B. Tovar, and S. Hutchinson. A sampling-based motion planning approach to maintain visibility of unpredictable targets. *Autonomous Robots*, 19(3):285–300, 2005.

[58] N. Noori and V. Isler. Lion and man with visibility in monotone polygons. In *Proc. Workshop on the Algorithmic Foundations of Robotics*, volume 86 of *Springer Tracts in Advanced Robotics*, pages 263–278. Springer, 2012.

[59] N. Noori and V. Isler. Lion and man game on convex terrains. In *Proc. Workshop on the Algorithmic Foundations of Robotics*, 2014.

[60] J. M. O'Kane. Decentralized tracking of indistinguishable targets using low-resolution sensors. In *Proc. IEEE International Conference on Robotics and Automation*, 2011.

[61] J. M. O'Kane and S. M. LaValle. On comparing the power of robots. *International Journal of Robotics Research*, 27(1):5–23, January 2008.

[62] J. M. O'Kane and W. Xu. Network-assisted target tracking via smart local routing. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2010.

[63] Jason M. O'Kane and Wenyuan Xu. Energy-efficient information routing in sensor networks for robotic target tracking. *Wireless Networks*, 18(6):713–733, 2012.

[64] J. O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, New York, NY, 1987.

[65] T. Ozaslan, S. Shen, Y. Mulgaonkar, N. Michael, and V. Kumar. Inspection of penstocks and featureless tunnel-like environments using micro uavs. In *Proc. International Conference on Field and Service Robotics*, 2013.

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

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

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

[69] N. N. Petrov. The cossack-robber differential game. *Differentsial'nye Uraveniya (Differential Equations)*, 19:1366–1374, 1983.

[70] J. Pita, M. Jain, J. Marecki, F. Ordó nez, C. Portway, M. Tambe, C. Western, P. Paruchuri, and S. Kraus. Deployed armor protection: The application of a game theoretic model for security at the los angeles international airport. In *Proc. International Conference on Autonomous Agents and Multiagent Systems*, pages 125–132, 2008.

[71] M. Pontani and B. A. Conway. Optimal interception of evasive missile warheads: Numerical solution of the differential game. *Journal of Guidance, Control, and Dynamics*, 31(4):1111–1122, 2008.

[72] M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

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

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

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

[76] J. T. Schwartz and M. Sharir. On the Piano Movers' Problem: II. General techniques for computing topological properties of algebraic manifolds. *Advances in Applied Mathematics*, 4(3):298–351, 1983.

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

[78] E. Shieh, B. An, R. Yang, M. Tambe, C. Baldwin, J. Direnzo, B. Maule, and G. Meyer. Protect: A deployed game theoretic system to protect the ports of the united states. In *Proc. International Conference on Autonomous Agents and Multiagent Systems*, 2012.

[79] N. Shrivastava, R. Mudumbai U. Madhow, and S. Suri. Target tracking with binary proximity sensors: fundamental limits, minimal descriptions, and algorithms. In *Proc. International Conference on Embedded Networked Sensor Systems*, pages 251–264, 2006.

[80] G. Simon, M. Maróti, Á. Lédeczi, G. Balogh, B. Kusy, A. Nádas, G. Pap, J. Sallai, and K. Frampton. Sensor network-based countersniper system. In *Proc. International Conference on Embedded Networked Sensor Systems*, pages 1–12. ACM, 2004.

[81] R. N. Smith, M. Schwager, S. L. Smith, B. H. Jones, D. Rus, and G. S. Sukhatme. Persistent ocean monitoring with underwater gliders: Adapting spatiotemporal sampling resolution. *Journal of Field Robotics*, 28(5):714–741, September 2011.

[82] S. L. Smith, M. Schwager, and D. Rus. Persistent robotic tasks: Monitoring and sweeping in changing environments. *IEEE Transactions on Robotics*, 28(2):410–426, 2012.

[83] N. M. Stiffler and J. M. O'Kane. Visibility-based pursuit-evasion with probabilistic evader models. In *Proc. IEEE International Conference on Robotics and Automation*, pages 4254–4259. IEEE, 2011.

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

[85] N. M. Stiffler and J. M. O'Kane. A complete algorithm for visibility-based pursuit-evasion with multiple pursuers. In *Proc. IEEE International Conference on Robotics and Automation*, pages 1660–1667, 2014.

[86] N. M. Stiffler and J. M. O'Kane. A sampling based algorithm for multi-robot visibility-based pursuit-evasion. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1782–1789, 2014.

[87] N. M. Stiffler and J. M. O'Kane. Pursuit-evasion with fixed beams. In *Proc. IEEE International Conference on Robotics and Automation*, page To Appear, 2016.

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

[89] T. V. T. V. Abramovskaya and N. N. Petrov. The theory of guaranteed search on graphs. *Vestnik St. Petersburg University: Mathematics*, 46(2):49–75, 2013.

[90] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics.* MIT Press, Cambridge, MA, 2005.

[91] P. Tokekar and V. Kumar. Visibility-based persistent monitoring with robot teams. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2015.

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

[93] J. Vander Hook and V. Isler. Pursuit and evasion with uncertain bearing measurements. In *Proc. Candadian Conference on Computational Geometry*, 2014.

[94] V. V. Vazirani. *Approximation Algorithms.* Springer-Verlag New York, Inc., New York, NY, 2001.

[95] D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms.* Cambridge University Press, New York, NY, USA, 1 edition, 2011.

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

[97] F. Zhao, J. Shin, and J. Reich. Information-driven dynamic sensor collaboration. *IEEE Signal Processing Magazine*, 19(2):61–72, 2002.

Appendix A

Cylindrical algebraic decomposition

Definition 26. A *cylindrical decomposition* of $\mathbb{R}^n$ is a partition of the space into cells that are constructible sets, such that the cells in the partition are cylindrically arranged.

This means the projection of any two cells onto any lower dimensional space are either equal or disjoint.

Definition 27. A *semi-algebraic decomposition* is a partition of $\mathbb{R}^n$ over a set of polynomials into a finite set of disjoint connected regions that are each sign invariant.[1]

Figure A.1 shows a sample environment and the corresponding semi-algebraic decomposition.

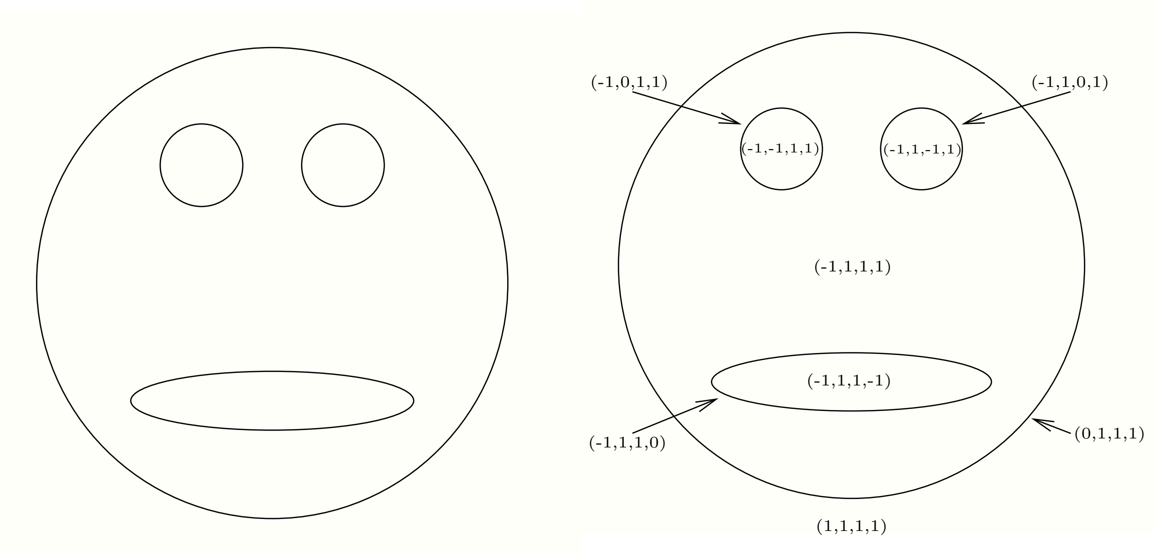
Definition 28. A *cylindrical algebraic decomposition* (CAD) [18] is a cylindrical semi-algebraic decomposition.

Collins [18] is the original developer of CAD, and provided an algorithm that takes as input a collection of polynomials in $\mathbb{Q}[x_1 \ldots x_n]$ and constructs a sign invariant CAD of $\mathbb{R}^n$.

CAD was originally designed to solve the quantifier elimination problem, but with the advent of a cell adjacency test [2], CAD could be effectively used in other domains,

[1] This means that inside each region, the sign for each polynomial remains constant (negative, zero, positive).

notably motion planning [46, 47, 76]. Figure A.2 shows the CAD and accompanying adjacency graph for the Gingerbread Face from Figure A.1.

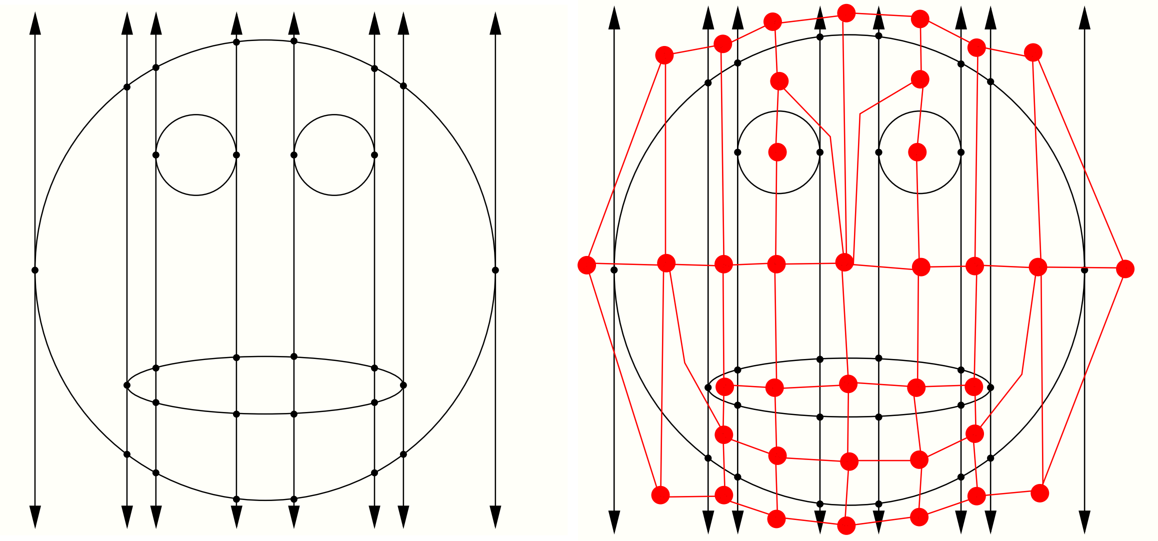


Gingerbread Face

Semi-algebraic decomposition

Steve LaValle, Planning Algorithms, 2006.

Figure A.1: An environment described by four polynomials and it's semi-algebraic decomposition.



CAD

Adjacency Graph

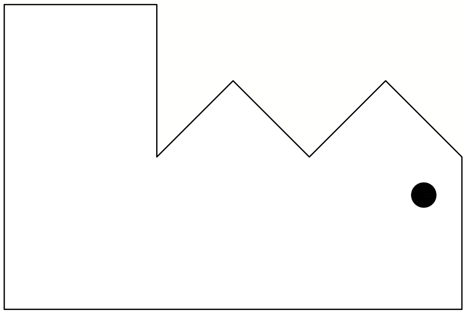
Steve LaValle, Planning Algorithms, 2006.

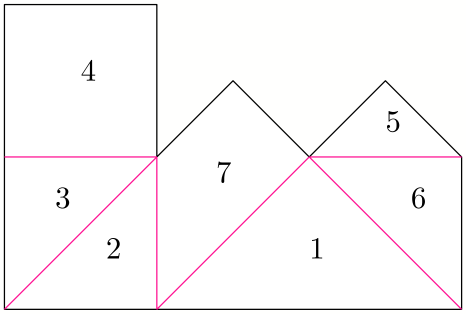
Figure A.2: The CAD of the gingerbread face and the adjacency graph corresponding to the transitions that exists between cells.

Appendix B

Single Pursuer - Optimal Shortest Path Forward Search Example

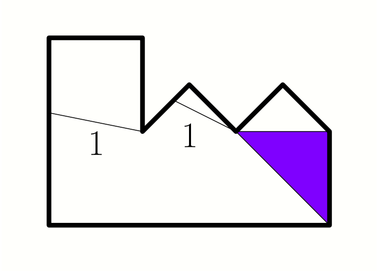
This appendix presents a detailed execution example for Algorithm 3. The intent is to illustrate, step by step, how the forward search progresses from a start node to a goal node (and by extension a motion strategy for the pursuer). At each iteration the contents of the priority queue are displayed at the top of the page. The expansion of the head node occurs in the center of the page. The list of non-dominated sequences appears at the bottom of the page.



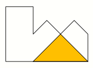
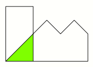
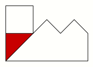
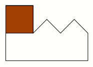
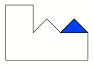
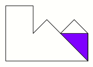
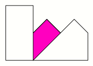


Start Node

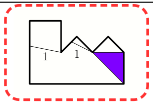
Region 6 - Fully Contaminated



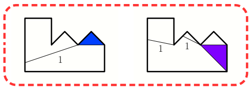
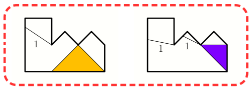
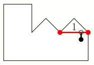
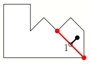
Non-dominated sequences

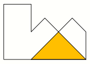
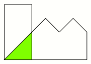
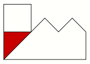
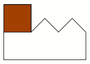
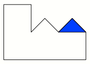
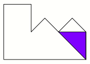
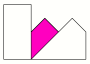
–Priority Queue–

Sequence	
Cost	0

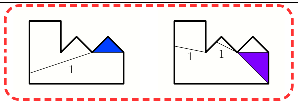
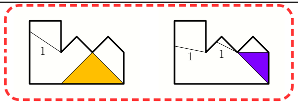
–Expansion(s)–

Sequence		
Tour		
Cost	.5	.707
Action	Add	Add

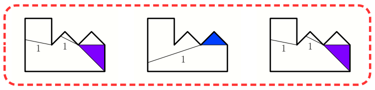
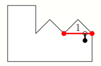
–Non-dominated Sequences–

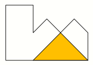
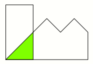
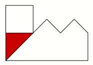
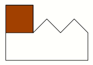
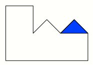
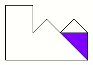
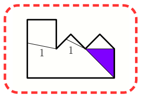
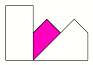
–Priority Queue–

Sequence		
Cost	.5	.707

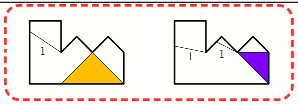
–Expansion(s)–

Sequence	
Tour	
Cost	.5
Action	Prunable – Omit

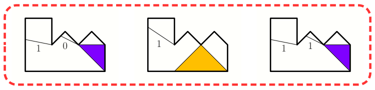
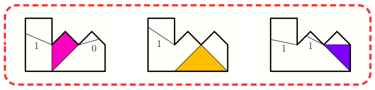
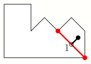
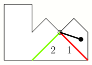
Non-dominated Sequences

	
	
	
	
	
	0 
	

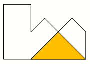
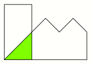
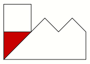
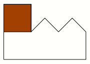
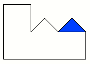
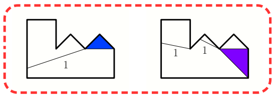
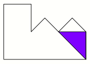
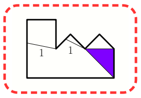
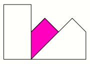
–Priority Queue–

Sequence	
Cost	.707

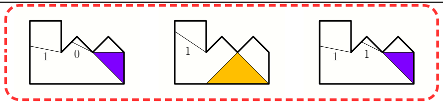
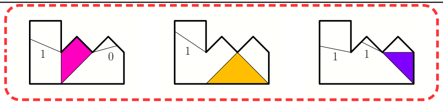
–Expansion(s)–

Sequence		
Tour		
Cost	.707	1.581
Action	Add	Add

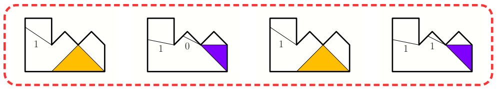
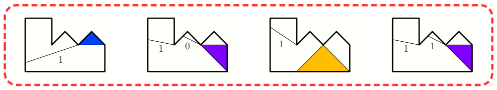
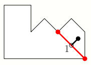
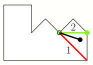
Non-dominated Sequences

	
	
	
	
	.5 
	0 
	

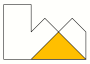
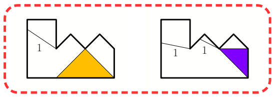
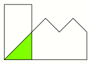
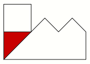
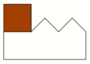
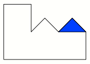
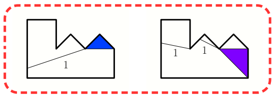
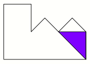
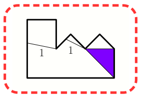
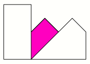
–Priority Queue–

Sequence		
Cost	.707	1.581

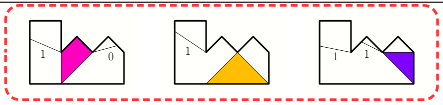
–Expansion(s)–

Sequence		
Tour		
Cost	.707	1.581
Action	Prunable – Omit	Prunable – Omit

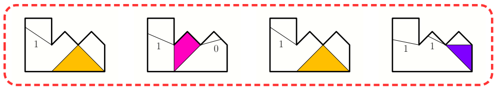
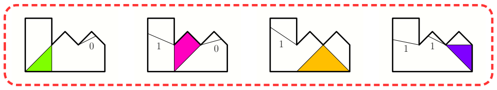
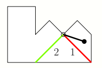
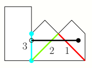
Non-dominated Sequences

	.707 
	
	
	
	.5 
	0 
	

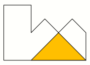
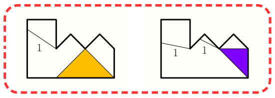
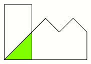
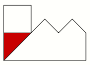
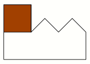
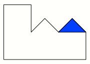
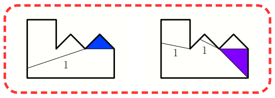
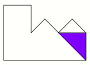
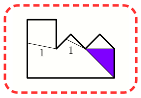
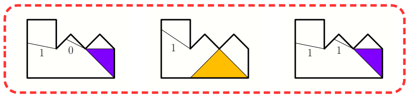
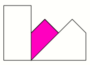
–Priority Queue–

Sequence	
Cost	1.581

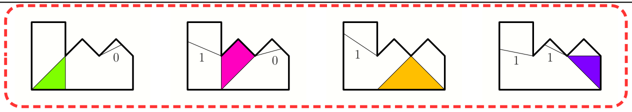
–Expansion(s)–

Sequence		
Tour		
Cost	1.581	3.5
Action	Prunable – Omit	Add

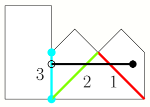
Non-dominated Sequences

	.707 
	
	
	
	.5 
	0  .707 
	

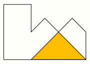
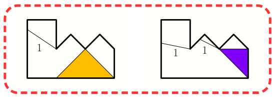
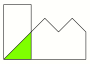
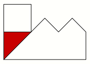
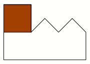
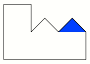
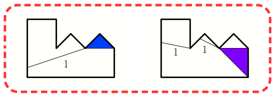
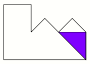
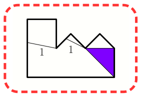
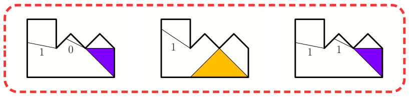
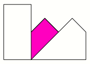
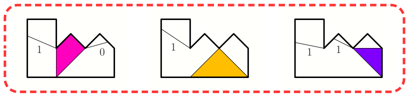
–Priority Queue–

Sequence	
Cost	3.5

Solution Found!!!



Non-dominated Sequences

	.707 
	
	
	
	.5 
	0  .707 
	1.581 

List of Algorithms