UNIVERSITY of CALIFORNIA
Santa Barbara

# Visibility Problems for Sensor Networks and Unmanned Air Vehicles

A Dissertation submitted in partial satisfaction of the
requirements for the degree

Doctor of Philosophy
in
Mechanical Engineering

by

Karl J. Obermeyer

Committee in charge:

Professor Francesco Bullo, Chair
Professor Bassam Bamieh
Professor Jeff Moehlis
Professor João P. Hespanha
Professor Subhash Suri

June 2010

Visibility Problems for Sensor Networks and Unmanned Air Vehicles

Copyright © 2010

by

Karl J. Obermeyer

*Meinem Vater, dem Fritz, der Nome und meinem Opa gewidmet*

# Acknowledgements

Foremost I must thank my advisor, Prof. Francesco Bullo, whose incredible energy, enthusiasm, optimism, and wisdom make him one of a kind. Coming to UCSB to work with him has proven a great decision. Thank you to Prof. Todd Murphey for recommending Prof. Bullo to me while I was still applying to graduate schools. Thank you also to the other members of my doctoral committee, Professors Bassam Bamieh, João Hespanha, Jeff Moehlis, and Subhash Suri, for their service and valuable feedback. I am also grateful to the UCSB Mechanical Engineering administrative staff, especially Laura Reynolds.

When I was an undergraduate at CU Boulder, I had the pleasure of conducting my first research projects with Prof. James Curry, Prof. James Meiss, and senior graduate student (now Dr.) Derin Wysham under the Applied Math Department's NSF VIGRE (Vertical Integration of Research and Education) grant. This experience introduced me to the research process and prepared me for my Ph.D. studies in a way that seems indispensable in retrospect. My sincere thanks to Prof. Curry, Prof. Meiss, Derin Wysham, the CU Boulder Applied Math Department, and the NSF.

Anurag Ganguli, Ketan Savla, and Sara Susca, who were senior graduate students during my beginning years at UCSB, deserve many thanks. I benefitted a great deal from their advice and perspectives on various control and robotics topics. I thank Anurag additionally for the collaboration on much of the work in this dissertation. Thanks are due also to my other academic siblings and cousins for their friendship and great research discussions, particularly Shaunak Bopardikar, Florian Dörfler, Joey Durham, Michael Schuresko, and Steve Smith.

Thanks to my mathematician friends Jordan Fisher, Rick Spjut, and Dave Valdman for lending their perspectives on my research and being available for analysis and topology consultations. Thanks also to Prof. Mihai Putinar for all of his office hours I spent learning measure theory.

This research was supported by a US Department of Defense SMART Fellowship for which I am grateful. As part of the SMART program, I spent three summers interning at the Control Design and Analysis Branch of AFRL, Wright-Patterson Air Force Base. For insightful interactions and collaboration I thank the many people I met directly or indirectly through AFRL, especially Maruthi Akella, Nicola Ceccarelli, Swaroop Darbha, Raymond Holsapple, Derek Kingston, Mark Mears, James Myatt, Paul Oberlin, Steve Rasmussen, Corey Schumacher, Vitaly Shaferman, and Tyler Summers.

Software implementation of control and planning algorithms is an important part of robotics research. When I first began PhD studies, my programming skills were weak. Securing my theoretical contributions was demanding enough that I struggled to find time for learning to program adequately. I owe a great debt of gratitude to my brother Fritz Obermeyer for tutoring me in the ways of software development, and for all the occasions when he somehow magically could see bugs in my code without even knowing what algorithm I was implementing. Thanks also to Donald Knuth for the LaTeX software used to typeset all my publications, and to the Free Software Foundation for Emacs and the gcc compiler used for my simulations.

Last but not least, I would like to thank the rest of my family and extended family, including Mary Dinh and Shaw Lynds, for their support through this research marathon.

# Curriculum Vitæ

Karl J. Obermeyer

## Education

| | |
|---|---|
| 2001–2005 | MS and BS in Applied Mathematics, University of Colorado at Boulder, Boulder, CO, USA. |

## Experience

| | |
|---|---|
| 2007–2009 | Summer Intern, Control Design and Analysis Branch of US Air Force Research Lab, Wright-Patterson Air Force Base, Dayton, OH, USA. |
| 2002–2005 | Project Engineer, Obermeyer Hydro Inc., Wellington, CO, USA. |
| 1998–1999 | Machinist, Perry Technology Corp., New Hartford, CT, USA. |

## Selected Publications

K. J. Obermeyer, A. Ganguli, and F. Bullo, "Multi-Agent Deployment for Visibility Coverage in Polygonal Environments with Holes", Note: Journal Article In Preparation.

K. J. Obermeyer, P. Oberlin, and S. Darbha, "Sampling-Based Roadmap Methods for a Visual Reconnaissance UAV", in *AIAA Journal of Guidance, Control, and Dynamics*, 2010, Note: Under Review.

(continued)

K. J. Obermeyer, A. Ganguli, and F. Bullo, "A Complete Algorithm for Searchlight Scheduling", in *International Journal of Computational Geometry and Applications*, 2010, Note: Under Review.

K. J. Obermeyer, "Path Planning for a UAV Performing Reconnaissance of Static Ground Targets in Terrain", in *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, Chicago, 2009.

K. J. Obermeyer, A. Ganguli, and F. Bullo, "Asynchronous Distributed Searchlight Scheduling", in *Proceedings of IEEE Conference on Decision and Control*, New Orleans, 2007.

K. J. Obermeyer and Contributors, "The VisiLibity Library: A C++ library for floating-point visibility computations", at `http://www.VisiLibity.org`, 2008.

**Abstract**

Visibility Problems for Sensor Networks and Unmanned Air Vehicles

by

Karl J. Obermeyer

This dissertation presents novel motion coordination, planning, and control algorithms to solve *visibility problems* for mobile sensor networks and UAVs (Unmanned Air Vehicles). These are problems where an autonomous system must move in such a way that it achieves or maintains line of sight of some object(s) of interest in a nonconvex environment. More specifically, we address variations of the following three problems: (1) How should a group of camera-equipped robotic agents deploy into an environment in order to achieve full visibility of that environment, (2) how can the rotations of security cameras be coordinated to ensure an intruder is caught on video, and (3) what path should a UAV follow in order to photograph a set of suspicious vehicles in a city as quickly as possible? We find answers to these questions using a unique blend of tools from the research domains of computational geometry, combinatorics, robot motion planning, and control theory.

# Contents

# List of Figures

# Chapter 1

# Introduction

Due to advances in mechanical, electrical, and computer engineering, the last half century has witnessed the inception of *autonomous systems* in the form of robots, networks of sensors and actuators, autonomous vehicles, and other mechatronic devices. These systems are increasingly being used in both civilian and military applications which are difficult, dangerous, or impossible for humans, e.g., environmental monitoring, geological survey, surgery, surveillance, reconnaissance, and search and rescue. Good coordination, planning, and control algorithms are a key component of autonomous systems technology because they can increase operational capabilities while reducing risk, costs, and operator workloads. This dissertation presents novel motion coordination, planning, and control algorithms to solve *visibility problems* for mobile sensor networks and UAVs (Unmanned Air Vehicles). These are problems where an autonomous system must move in such a way that it achieves or maintains line of sight of some object(s) of interest in a nonconvex environment. An objects of interest could be, e.g., a robot, a vehicle, a human, an animal, or the environment itself. More specifically, we address variations of the following three problems.

**Distributed Visibility-Based Deployment Problem with Connectivity:** Design a distributed algorithm for a network of autonomous camera-equipped robotic agents to deploy into an unmapped nonconvex polygonal environment such that (1) they maintain a line-of-sight connected network, and (2) from their final positions every point in the environment is visible to some agent. The agents are to begin deployment from a common point and operate using only information from local sensing and line-of-sight communication.

**Seachlight Scheduling Problem:** Find a schedule to rotate a set of searchlights (modeled as rays) or cameras with limited field of view (modeled as cones) such that any intruder in a nonconvex polygonal environment will necessarily be detected in finite time.

**Reconnaissance Path Planning Problem for a UAV:** Given a set of stationary ground targets in a terrain (natural, urban, or mixed), compute a flyable path for a camera-equipped UAV such that it can photograph all targets in minimum time.

Traditional motion coordination, planning, and control has been concerned primarily with stearing an autonomous system between states such that (1) dynamic constraints are satisfied, and (2) collisions are avoided. The key difficulty in visibility problems is that not only the same two constraints must be satisfied as in the traditional setting, but also special line-of-sight geometric constraints must be satisfied.

## 1.1 Relevant Literature

We give here an overview of some literature involving visibility problems. More problem-specific literature reviews can be found in the individual chapters.

### Visibility Coverage

The Distributed Visibility-Based Deployment Problem with Connectivity is a visibility coverage problem. Approaches to visibility coverage can be divided into two categories: those where the environment is known a priori and those where the environment must be discovered. When the environment is known a priori, a well-known approach is the *Art Gallery Problem* in which one seeks the smallest set of guards such that every point in a polygon is visible to some guard. This problem has been shown both NP-hard [1] and APX-hard [2] in the number of vertices $n$ representing the environment.[1] The best known approximation algorithms offer solutions only within a factor of $O(\log g)$, where $g$ is the optimum number of agents [4]. The *Art Gallery Problem with Connectivity* is the same as the Art Gallery Problem, but with the additional constraint that the guards' visibility graph must consist of a single connected component, i.e., the guards must form a connected network by line of sight. This problem is also NP-hard in $n$ [5]. Many other variations on the Art Gallery Problem are well surveyed in [6, 7, 8]. The classical *Art Gallery Theorem*, proven first in [9] by induction and in [10] by a beautiful coloring argument, states that $\lfloor \frac{n}{3} \rfloor$ vertex guards[2] are always sufficient and sometimes necessary to cover a polygon with $n$ vertices and no holes. The *Art Gallery Theorem with Holes*, later proven independently by [11] and [12],

[1] Definitions of computational complexity classes such as NP can be found, e.g., in [3].

[2] A *vertex guard* is a guard which is located at a vertex of the polygonal environment.

states that $\lfloor \frac{n+h}{3} \rfloor$ point guards[3] are always sufficient and sometimes necessary to cover a polygon with $n$ vertices and $h$ holes. If guard connectivity is required, [13] proved by induction and [14] by a coloring argument, that $\lfloor \frac{n-2}{2} \rfloor$ vertex guards are always sufficient and occasionally necessary for polygons without holes. We are not aware of any such bound for connected coverage of polygons with holes. For polygonal environments with holes, centralized camera-placement algorithms described in [15] and [16] take into account practical imaging limitations such as camera range and angle-of-incidence, but at the expense of being able to obtain worst-case bounds as in the Art Gallery Theorems. The constructive proofs of the Art Gallery Theorems rely on global knowledge of the environment and thus are not amenable to emulation by distributed algorithms.

One approach to visibiliy coverage when the environment must be discovered is to first use SLAM (Simultaneous Localization And Mapping) techniques [17, 18] to explore and build a map of the entire environment, then use a centralized procedure to decide where to send agents. In [19], for example, deployment locations are chosen by a human user after an initial map has been built. Waiting for a complete map of the entire environment to be built before placing agents may not be desirable. In [20] agents fuse sensor data to build only a map of the portion of the environment covered so far, then heuristics are used to deploy agents onto the frontier of the this map, thus repeating this procedure incrementally expands the covered region. For any techniques relying heavily on SLAM, however, synchronization and data fusion can pose significant challenges under communication bandwidth limitations. In [21] agents discover and achieve visibility coverage of an environment not by building a geometric map, but instead by sharing only

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

combinatorial information about the environment; however, the strategy focuses on the theoretical limits of what can be achieved with minimalistic sensing, thus the amount of robot motion required becomes impractical.

Most relevant to, and a source of inspiration for the work in Chapter 2, are the distributed visibility-based deployment algorithms, for polygonal environments without holes, developed recently by Ganguli et al [22, 23, 24]. These algorithms are simple, require only limited impact-based communication, and offer worst-case optimal bounds on the number of agents required. The basic strategy is to incrementally construct a so-called *nagivation tree* through the environment. To each vertex in the navigation tree corresponds a region of the the environment which is completely visible from that vertex. As agents move through the environment, they eventually settle on certain nodes of the navigation tree such that the entire environment is covered.

## Visibility-Based Pursuit-Evasion

Seachlight Scheduling is a form of *visibility-based pursuit-evasion*, that is, pursuit-evasion where the goal is to see an evader rather than achieve physical proximity to it. To our knowledge the Searchlight Scheduling Problem was first introduced by Sugihara, Suzuki and Yamashita [25]. They give a solution, the "One Way Sweep Strategy", to the limited class of searchlight scheduling problem instances in which the environment is simply connected and there is at least one searchlight located on the boundary for every connected component of their visibility graph. In [26] an upper bound is given on the number of guards with multiple searchlights sufficient in polygonal environments containing holes. We adopt the convention in [26] and call a mobile guard possessing $k$ searchlights a *$k$-searcher*. Some articles involving

1-searchers, sometimes calling them *flashlights* or *beam detectors*, are [27], [28], [29], and [30]. Other treatments of visibility-based pursuit evasion problems in simple 2D environments include, e.g., [31, 32, 33, 34, 35, 36, 37, 38, 39, 40].

Exact cell decomposition, a method we use in Chapter 3, has been used in the design of complete algorithms to solve visibility-based pursuit-evasion problems before, e.g., in [32] and [27]. In [32] an algorithm is given for a single mobile searcher with omnidirectional vision, and it is shown that determining the minimum number of such pursuers required to clear a polygonal environment with holes is NP-hard. In [27] a complete algorithm is described for a single mobile "$\phi$-searcher" having an angle $\phi$ field of view, and it is shown that determining the minimum number of such pursuers required to clear a polygonal environment with holes is also NP-hard. To our knowledge nobody has carried out the design of a complete algorithm to solve any visibility-based pursuit-evasion problem involving arbitrary polygonal environments with holes. However, there are at least two noteworthy articles involving multiple pursuers in polygonal environments without holes. In [41] a polynomial time complete algorithm is provided for two 1-searchers in a simple polygonal environment, but has not been extended to three or greater pursuers and it is not clear how to do so. In [42] a polynomial time complete algorithm is given to determine the minimum number of $\infty$-searchers (omnidirectional vision) necessary to clear a simple polygon, but under the constraints that (1) the pursuers are in a chain configuration where consecutive pursuers along the chain are mutually visible, and (2) end pursuers must remain on the polygon boundary.

## Reconnaissance Path Planning for a UAV

UAVs are becoming the sensor platform of choice for many surveillance, reconnaissance, and search and rescue applications [43, 44, 45, 46]. The Reconnaissance Path Planning Problem which we treat in this dissertation is for a fixed-wing UAV modeled as a Dubins vehicle[4]. If we approximate as polygons the subsets of airspace from which targets are visible to a UAV, then our reconnaissance path planning problem is reduced to a Polygon-Visiting Dubins Traveling Salesman Problem (PVDTSP hereinafter). To our knowledge the PVDTSP has not previously been studied. Because the PVDTSP has embedded in it the combinatorial problem of choosing the order to visit the polygons, the solution space is very large and discontinuous. This precludes direct application of numerical optimal control techniques traditionally used in trajectory optimization, surveyed, e.g., in [47]. However, several related variations of the TSP are of interest. The *ETSP (Euclidean TSP)* is a TSP where the vertices of the graph are points in the Euclidean plane $\mathbb{R}^2$ and the edges are weighted with Euclidean distances. In the *ETSPN (Euclidean TSP with Neighborhoods)* one seeks a shortest closed Euclidean path passing through $n$ subsets of the plane. The ETSP is NP-hard [48] and so is the ETSPN by virtue of being a generalization of the ETSP. The *DTSP (Dubins TSP)*, where a Dubins vehicle must follow a shortest tour through $n$ single point targets in the plane, is known to be NP-hard in $n$ [49]. Various heuristics for both single and multi-vehicle versions of the DTSP can be found, e.g., in [50], [51], and [52]. The PVDTSP reduces to the ETSPN in the limit as the vehicle's minimum turning radius becomes small compared to the distances between polygons. Similarly, as the area of the polygons goes to zero, the PVDTSP re-

[4] A Dubins vehicle is one which moves only forward and has a minimum turning radius [45, 46].

duces to the DTSP, hence the PVDTSP is NP-hard. There exist a number of algorithms with approximation guarantees for both the DTSP [53, 54, 55] and ETSPN [56, 57, 58], but it appears that extending any of these algorithms to the PVDTSP would put undesirable restirictions on the problem instances which could be handled, e.g., the polygons would not be allowed to overlap. The *FOTSP (Finite One-in-set TSP)*[5] is the problem of finding a closed path of minimum cost which passes through at least one vertex in each of a finite collection of *clusters*, the clusters being mutually exclusive finite vertex sets. The FOTSP is NP-hard because it has as a special case the *ATSP (Asymmetric TSP)* [59]. An FOTSP instance can be solved exactly by transforming it into an ATSP instance using the *Noon-Bean transformation* from [60], then invoking an ATSP solver. Alternatively, an FOTSP can be solved using an approximate dynamic programming technique as in [61]. In the robotics literature [18, 62], a *sampling-based roadmap method*[6] refers to any algorithm which operates by sampling a finite set of points from a continuous state space in order to reduce a continuous motion planning problem to planning on a finite discrete graph. Sampling-based roadmap methods have traditionally only been used for collision-free point-to-point path planning amongst obstacles, however, in [63] approximate solutions to the DTSP are found by sampling discrete sets of orientations that the Dubins vehicle can have over each target, essentially approximating a DTSP instance by an FOTSP instance. They then use the Noon-Bean transformation to convert the FOTSP instance into an ATSP instance so that a standard ATSP solver can be applied. Discretization of the vehicle state space in order to approximate the original problem by an

[5]What we have chosen to call the FOTSP is known variously in the literature as "Group-TSP", "Generalized-TSP", "One-of-a-Set TSP", "Errand Scheduling Problem", "Multiple Choice TSP", "Covering Salesman Problem", or "International TSP".

[6]In this usage, "method" means a high level algorithm having multiple components, each of which may be considered an algorithm in its own right.

FOTSP is a key idea which we build upon in designing sampling-based roadmap methods for the PVDTSP in the present work. For NP-hard problems such as the TSP and most of its variations, another possible approach is to use meta-heuristic algorithms, e.g., tabu search, simulated annealing, or genetic algorithms [64]. These techniques typically lack performance guarantees, yet obtain good solutions in reasonable computation time. Particularly, genetic algorithms have recently been applied to variations of the TSP and UAV motion planning problems [65, 66, 67, 68, 69, 70, 71]. *Genetic algorithm* is an umbrella term referring to any iterative procedure which mimics biological evolution by operating on a population of candidate solutions encoded as so-called chromosomes. The genetic operators of crossover and mutation are successively applied, generation after generation, until a sufficiently fit solution appears in the population. It is not obvious how to adapt existing genetic algorithms to the PVDTSP, nor is it clear whether any such adaptations would be effective.

## Tracking, Exploration, and Visibility in Other Disciplines

There are several other research areas involving visibility which are certainly worth mentioning despite being less relevant to the present work than most references given so far. The problem of tracking a moving target while maintaining line of sight in the presence of occlusions is treated, e.g., in [72, 73, 70, 71, 74]. Robotic exploration for the purpose of map building has been the subject of extensive research [18, 17], but usually statistical rather than combinatorial methods are used. The field of computer vision [75, 76] focuses on pure imaging and sensing issues rather than motion control, coordination, or planning. In computer graphics and computational geometry, visibility problems focus on efficient computation of

visibility sets and occlusion from geometric environment models, e.g., see [77, 74] and references therein.

## 1.2 Organization and Contributions

The organization and contributions of this dissertation are summarized as follows.

**Chapter 2: Multi-Agent Deployment for Visibility Coverage**

The contribution of this chapter is the design of an algorithm which solves the Distributed Visibility-Based Deployment Problem with Connectivity in polygonal environments with holes. For this algorithm we prove (i) convergence, (ii) worst-case upper bounds on the time and number of agents required, (iii) bounds on the memory and communication complexity, and (iv) robust extensions. Simulation results are also included. Our algorithm operates using line-of-sight communication and by use of a so-called *partition tree* distributed data structure similar to the *navigation tree* used by Ganguli et al as described above. In polygonal environments with holes the algorithms of Ganguli et al fail because branches of the navigation tree conflict when they wrap around one or more holes. Our algorithm, however, is able to handle such "branch conflicts".

**Chapter 3: Centralized Searchlight Scheduling**

There are three main contributions in this chapter. First, we show by exact cell decomposition that if an instance of the Searchlight Scheduling Problem permits any solution at all, then it also permits a solution in a reduced discrete solution space. The second contribution is to use the knowledge of

the solution space discretization to design a complete[7] algorithm for searchlight scheduling. Although it remains an open problem whether searchlight scheduling is NP-hard, our computed examples demonstrate that for searchlights, even in environments with holes, the time complexity of a complete exact cell decomposition is not entirely prohibitive and can be practical for problem instances of useful size. At this time no other algorithm exists to solve the general Searchlight Scheduling Problem. As a third contribution we treat a new problem which we call the *$\phi$-Searchlight Scheduling Problem* in which *$\phi$-searchlights* sense not just along a ray, but over a finite field of view (as for a typical security camera). We show how our searchlight scheduling algorithm can be extended to take advantage of $\phi$-searchlights having a wider field of view than just a ray. This is an important extension because for cameras having a finite field of view it is a much more realistic sensor model. We envision our algorithms and/or other algorithms inspired by this work will one day be used in automating the design of security systems consisting of networks of statically positioned rotating sensors and actuators.

**Chapter 4: Distributed Searchlight Scheduling**

The main contribution of this chapter is the development of two asynchronous distributed algorithms to solve the searchlight scheduling problem. Correctness and completion time bounds for nonconvex polygonal environments are discussed. The first algorithm, called DOWSS (Distributed One Way Sweep Strategy), is a distributed version of a known algorithm described originally in [25], but it can be very slow in clearing the entire

[7]Here *complete* means that if a solution exists, the algorithm is guaranteed to find one in finite time.

environment because only one searchlight may rotate at a time. On-line processing time required by agents during execution of DOWSS is relatively low, so that the expedience with which an environment can be cleared is essentially limited by the maximum angular speed searchlights may be rotated at. In an effort to reduce the time to clear the environment, we develop a second algorithm, called PTSS (Parallel Tree Sweep Strategy), which sweeps searchlights in parallel if guards are placed in appropriate locations. These locations are related to an environment partition with certain properties. That we analyze the time it takes to clear an environment, given a bound on the angular rotation velocity, is a unique feature among all work involving searchlights to date.

**Chapter 5: Path Planning for a Visual Reconnaissance UAV**

There are four main contributions in this chapter. First, we precisely formulate the general aircraft visual reconnaissance problem for static ground targets in terrain. Assuming the UAV can be modeled as a Dubins vehicle and that target visibility sets are polygons, we reduce our general formulation to a PVDTSP (Polygon-Visiting Dubins Traveling Salesman Problem). Although the PVDTSP reduces to the well-studied DTSP and ETSP in the *sparse limit* as targets are very far apart, we provide a worst-case analysis demonstrating the importance of developing specialized algorithms for the PVDTSP in the *dense limit* as targets are close together and polygons may overlap significantly. The second contribution is the design and numerical study of two sampling-based roadmap methods for the PVDTSP. These methods operate by sampling finite discrete sets of vehicle states to approximate a PVDTSP instance by an FOTSP instance, then applying

existing FOTSP solving techniques. One of our sampling-based roadmap methods uses the Noon-Bean transformation from [60] and is *resolution complete*, which means it provably converges to a nonisolated global optimum as the number of samples grows. Our other sampling-based roadmap method achieves faster computation times by using the approximate dynamic programming technique from [61], but consequently only converges to a nonisolated global optimum modulo target order. While we have borrowed the idea of approximation by an FOTSP from [63], the present work goes beyond a simple extension in that we (1) illustrate the connection with sampling-based roadmap methods used for path planning in the robotics literature[8], (2) use a novel sampling technique to reduce computational time complexity, and (3) provide proof of convergence to nonisolated global optima. The third contribution, is the design of a genetic algorithm for the PVDTSP. The genetic algorithm has no performance guarantees but is easiest to implement and tends to find good feasible solutions quickly. Numerical experiments indicate that both the sampling-based roadmap methods and genetic algorithm deliver good solutions suitably quickly for online purposes when applied to PVDTSP instances having up to about 20 targets. Additionally, all the algorithms have a means for a user to trade off computation time for solution quality. The fourth contribution is a description of how the modular nature of all the algorithms allows them to easily be extended to handle wind, airspace constraints, any vehicle dynamics, and open-path (vs. closed-tour) problems.

[8]Although [63] appears to be the first application of a sampling-based roadmap method to a TSP-type problem, they do not use the term "sampling-based roadmap method", nor is there any mention of the connection with sampling-based roadmap methods in the robotics literature.

# Chapter 2

# Multi-Agent Deployment for Visibility Coverage

## 2.1 Introduction

In this chapter we design a distributed algorithm for deploying a group of mobile robotic agents with omnidirectional vision into nonconvex polygonal environments with holes, e.g., an urban or building floor plan. Agents are identical except for their unique identifiers (UIDs), begin deployment from a common point, possess no prior knowledge of the environment, and operate only under line-of-sight sensing and communication. The objective of the deployment is for the agents to achieve full visibility coverage of the environment while maintaining line-of-sight connectivity (at any time the agents' visibility graph consists of a single connected component). We call this the *Distributed Visibility-Based Deployment Problem with Connectivity*. Once deployed, the agents may supply surveillance information to an operator through the ad-hoc line-of-sight communication network. A graphical description of our objective is given in Fig. 2.1.

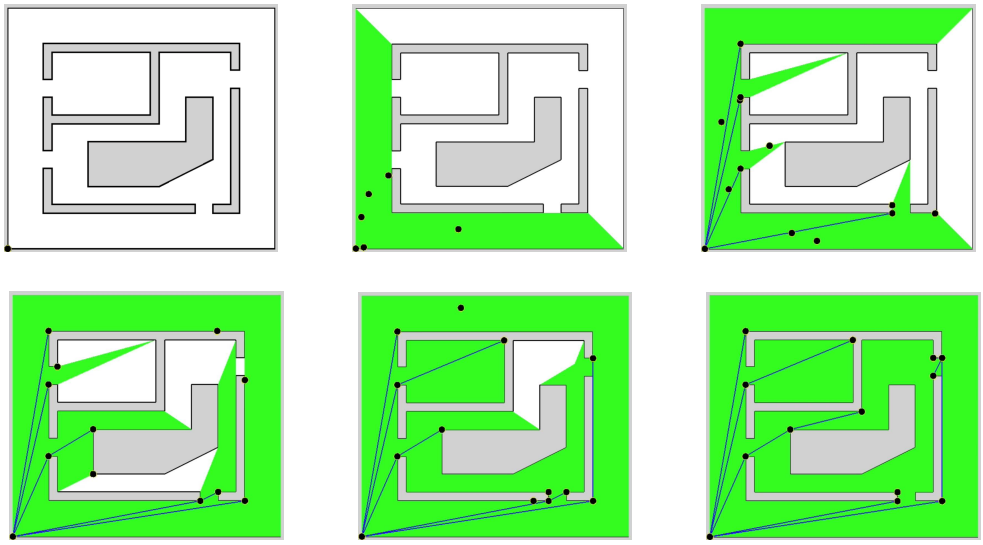


Figure 2.1. This sequence (left to right, top to bottom) shows a simulation run of the distributed visibility-based deployment algorithm described in Sec. 2.6. Agents (black disks) initially are colocated in the lower left corner of the environment. As the agents spread out, they claim areas of responsibility (green) which correspond to cells of the incremental partition tree $\mathcal{T}_{\mathcal{P}}$. Blue lines show line-of-sight connections between agents responsible for neighboring vertices of $\mathcal{T}_{\mathcal{P}}$. Once agents have settled to their final positions, every point in the environment is visibile to some agent and the agents form a line-of-sight connected network.

Approaches to visibility coverage problems can be divided into two categories: those where the environment is known a priori and those where the environment must be discovered. When the environment is known a priori, a well-known approach is the *Art Gallery Problem* in which one seeks the smallest set of guards such that every point in a polygon is visible to some guard. This problem has been shown both NP-hard [1] and APX-hard [2] in the number of vertices $n$ representing the environment. The best known approximation algorithms offer solutions only within a factor of $O(\log g)$, where $g$ is the optimum number of agents [4]. The *Art Gallery Problem with Connectivity* is the same as the Art

Gallery Problem, but with the additional constraint that the guards' visibility graph must consist of a single connected component, i.e., the guards must form a connected network by line of sight. This problem is also NP-hard in $n$ [5]. Many other variations on the Art Gallery Problem are well surveyed in [6, 7, 8]. The classical *Art Gallery Theorem*, proven first in [9] by induction and in [10] by a beautiful coloring argument, states that $\lfloor \frac{n}{3} \rfloor$ vertex guards[1] are always sufficient and sometimes necessary to cover a polygon with $n$ vertices and no holes. The *Art Gallery Theorem with Holes*, later proven independently by [11] and [12], states that $\lfloor \frac{n+h}{3} \rfloor$ point guards[2] are always sufficient and sometimes necessary to cover a polygon with $n$ vertices and $h$ holes. If guard connectivity is required, [13] proved by induction and [14] by a coloring argument, that $\lfloor \frac{n-2}{2} \rfloor$ vertex guards are always sufficient and occasionally necessary for polygons without holes. We are not aware of any such bound for connected coverage of polygons with holes. For polygonal environments with holes, centralized camera-placement algorithms described in [15] and [16] take into account practical imaging limitations such as camera range and angle-of-incidence, but at the expense of being able to obtain worst-case bounds as in the Art Gallery Theorems. The constructive proofs of the Art Gallery Theorems rely on global knowledge of the environment and thus are not amenable to emulation by distributed algorithms.

One approach to visibiliy coverage when the environment must be discovered is to first use SLAM (Simultaneous Localization And Mapping) techniques [17] to explore and build a map of the entire environment, then use a centralized procedure to decide where to send agents. In [19], for example, deployment locations

[1] A *vertex guard* is a guard which is located at a vertex of the polygonal environment.

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

are chosen by a human user after an initial map has been built. Waiting for a complete map of the entire environment to be built before placing agents may not be desirable. In [20] agents fuse sensor data to build only a map of the portion of the environment covered so far, then heuristics are used to deploy agents onto the frontier of the this map, thus repeating this procedure incrementally expands the covered region. For any techniques relying heavily on SLAM, however, synchronization and data fusion can pose significant challenges under communication bandwidth limitations. In [21] agents discover and achieve visibility coverage of an environment not by building a geometric map, but instead by sharing only combinatorial information about the environment, however, the strategy focuses on the theoretical limits of what can be achieved with minimalistic sensing, thus the amount of robot motion required becomes impractical.

Most relevant to and the inspiration for the present work are the distributed visibility-based deployment algorithms, for polygonal environments without holes, developed recently by Ganguli et al [22, 23, 24]. These algorithms are simple, require only limited impact-based communication, and offer worst-case optimal bounds on the number of agents required. The basic strategy is to incrementally construct a so-called *nagivation tree* through the environment. To each vertex in the navigation tree corresponds a region of the the environment which is completely visible from that vertex. As agents move through the environment, they eventually settle on certain nodes of the navigation tree such that the entire environment is covered.

The contribution of this chapter is the design of an algorithm which solves the Distributed Visibility-Based Deployment Problem with Connectivity in polygonal environments with holes. For this algorithm we prove (i) convergence, (ii)

worst-case upper bounds on the time and number of agents required, (iii) bounds on the memory and communication complexity, and (iv) robustness extensions. Simulation results are also included. Our algorithm operates using line-of-sight communication and a so-called *partition tree* data structure similar to the *navigation tree* used by Ganguli et al as described above. In polygonal environments with holes the algorithms of Ganguli et al fail because branches of the navigation tree conflict when they wrap around one or more holes. Our algorithm, however, is able to handle such "branch conflicts".

This chapter is organized as follows. We begin with some technical definitions in Sec. 2.2, then a precise statement of problem and assumptions in Sec. 2.3. Details on the agents' sensing, dynamics, and communication are given in Sec. 2.4. Algorithm descriptions, including pseudocode and simulation results, are presented in Sec. 2.5 and Sec. 2.6. We conclude in Section 2.7.

## 2.2 Notation and Preliminaries

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

in Table 2.4.

We turn our attention to the environment, visibility, and graph theoretic concepts. The environment $\mathcal{E}$ is polygonal with vertex set $V_\mathcal{E}$, edge set $E_\mathcal{E}$, total vertex count $n = |V_\mathcal{E}| = |E_\mathcal{E}|$, and hole count $h$. Given any polygon $c \subset \mathcal{E}$, the vertex set of $c$ is $V_c$ and the edge set is $E_c$. A segment $[a, b]$ is a *diagonal* of $\mathcal{E}$ if (i) $a$ and $b$ are vertices of $\mathcal{E}$, and (ii) $]a, b[ \subset \text{int}(\mathcal{E})$. Let $e$ be any point in $\mathcal{E}$. The point $e$ is *visible from* another point $e' \in \mathcal{E}$ if $[e, e'] \subset \mathcal{E}$. The *visibility polygon* $\mathcal{V}(e) \subset \mathcal{E}$ of $e$ is the set of points in $\mathcal{E}$ visible from $e$ (Fig. 2.2). The *vertex-limited visibility polygon* $\tilde{\mathcal{V}}(e) \subset \mathcal{V}$ is the visibility polygon $\mathcal{V}(e)$ modified by deleting every vertex which does not coincide with an environment vertex (Fig. 2.2). A *gap edge* of $\mathcal{V}(e)$ (resp. $\tilde{\mathcal{V}}(e)$) is defined as any line segment $[a, b]$ such that $]a, b[ \subset \text{int}(\mathcal{E})$, $[a, b] \subset \partial\mathcal{V}(e)$ (resp. $[a, b] \subset \partial\tilde{\mathcal{V}}(e)$), and it is maximal in the sense that $a, b \in \partial\mathcal{E}$. Note that a gap edge of $\tilde{\mathcal{V}}(e)$ is also a diagonal of $\mathcal{E}$. For short, we refer to the gap edges of $\mathcal{V}(e)$ as the *visibility gaps* of $e$. A set $R \subset \mathcal{E}$ is *star-convex* if there

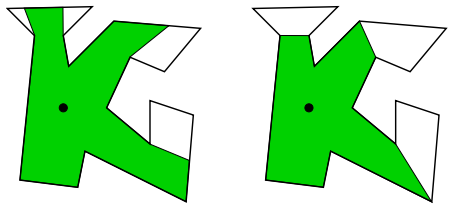


Figure 2.2. In a simple nonconvex polygonal environment are shown examples of the visibility polygon (red, left) of a point observer (black disk), and the vertex-limited visibility polygon (red, right) of the same point.

exists a point $e \in R$ such that $R \subset \mathcal{V}(e)$. The *kernel* of a star-convex set $R$, is the set $\{e \in \mathcal{E} | R \subset \mathcal{V}(e)\}$, i.e., all points in $R$ from which all of $R$ is visible. The *visibility graph* $\mathcal{G}_{\text{vis}}\mathcal{E}(P)$ of a set of points $P$ in environment $\mathcal{E}$ is the undirected graph with $P$ as the set of vertices and an edge between two vertices if and only

if they are (mutually) visible. A *tree* is a connected graph with no simple cycles. A *rooted tree* is a tree with a special vertex designated as the *root*. The *depth* of a vertex in a rooted tree is the minimum number of edges which must be treversed to reach the root from that vertex. Given a tree $\mathcal{T}$, $V_{\mathcal{T}}$ is its set of vertices and $E_{\mathcal{T}}$ its set of edges.

## 2.3 Problem Description and Assumptions

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

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

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

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

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

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

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

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

Later, in Sec. 2.6.6 we will describe how our nominal deployment algorithm can be extended to relax some assumptions.

## 2.4 Network of Visually-Guided Agents

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

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

$$\dot{p}^{[i]} = u^{[i]}, \tag{2.1}$$

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

is valid also for higher order dynamics.

The agents' communication graph is precisely their visibility graph $\mathcal{G}_{\text{vis}}\mathcal{E}(P)$, i.e., any *visibility neighbors* (mutually visible agents) may communicate with each other. Agents may send their messages using, e.g., UDP (User Datagram Protocol). Each agent ($i = 0, \ldots, N-1$) stores received messages in a FIFO (First-In-First-Out) buffer In_Buffer$^{[i]}$ until they can be processed. Messages are sent only upon the occurrence of certain asynchronous events and the agents' processors need not be synchronized, thus the agents form an *event-driven asynchronous robotic network* similar to that described, e.g., in [78]. In order for two visibility neighbors to establish a common reference frame, we assume agents are able to solve the *correspondence problem*: the ability to associate the messages they receive with the corresponding robots they can see. This may be accomplished, e.g., by the robots performing localization, however, as mentioned in Sec. 2.1, this might use up limited communication bandwidth and processing power. Simpler solutions include having agents display different colors, "license plates", or periodic patterns from LEDs [79].

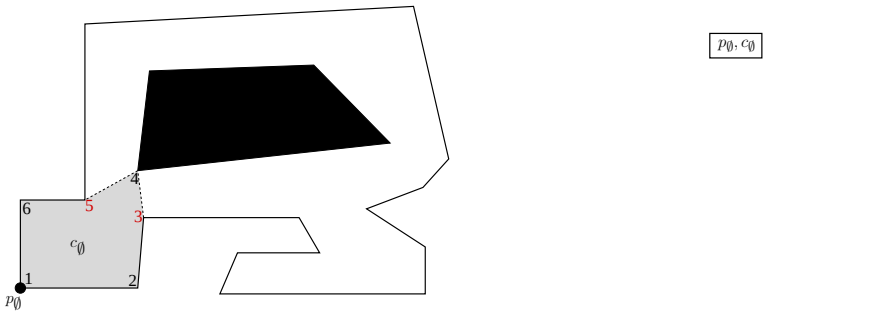
## 2.5 Incremental Partition Algorithm

We introduce a centralized algorithm to incrementally partition the environment $\mathcal{E}$ into a finite set of openly disjoint star-convex polygonal cells. Roughly, the algorithm operates by choosing at each step a new *vantage point* on the frontier of the uncovered region of the environment, then computing a cell to be covered by that vantage point (each vantage point is in the kernel of its corresponding cell). The frontier is pushed as more and more vantage point - cell pairs are added

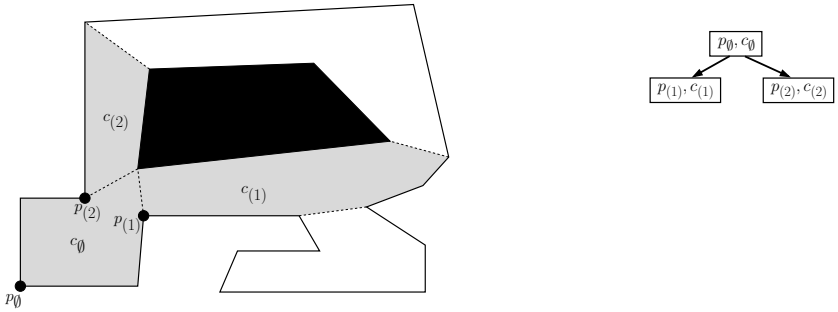
until eventually the entire environment is covered. The vantage point - cell pairs form a directed rooted tree structure called the *partition tree* $\mathcal{T}_{\mathcal{P}}$. This algorithm is a variation and extension of an incremental partition algorithm used in [24], the main differences being that we have added a protocol for handling holes and adapted the notation to better fit the added complexity of handling holes. The deployment algorithm to be described in Sec. 2.6 is a distributed emulation of the centralized incremental partition algorithm we present here.

Before examining the precise pseudocode Table 2.1, we informally step through the incremental partition algorithm for the simple example of Fig. 2.3a-f. This sequence shows the environment partition together with corresponding abstract representations of the partition tree $\mathcal{T}_{\mathcal{P}}$. Each vertex of $\mathcal{T}_{\mathcal{P}}$ is a vantage point - cell pair and edges are based on cell adjacency. Given any vertex of $\mathcal{T}_{\mathcal{P}}$, say $(p_\xi, c_\xi)$, $\xi$ is the *PTVUID (Partition Tree Vertex Unique IDentifier)*. The PTVUID of a vertex at depth $d$ is a $d$-tuple, e.g., (1), (2,1), or (1,1,1). The symbol $\emptyset$ is used as the root's PTVUID. The algorithm begins with the root vantage point $p_\emptyset$. The cell of $p_\emptyset$ is the grey shaded region $c_\emptyset$ in Fig. 2.3a, which is the vertex-limited visibility polygon $\tilde{\mathcal{V}}(p_\emptyset)$. According to certain technical criteria, made precise later, child vantage points are chosen on the endpoints of the unexplored gap edges. In Fig. 2.3a, dashed lines show the unexplored gap edges of $c_\emptyset$. Selecting $p_{(1)}$ as the next vantage point, the corresponding cell $c_{(1)}$ becomes the portion of $\tilde{\mathcal{V}}(p_{(1)})$ which is across the parent gap edge and extends away from the parent's cell. The vantage point $p_{(2)}$ and its cell $c_{(2)}$ are generated in the same way. There are now three vertices, $(p_\emptyset, c_\emptyset)$, $(p_{(1)}, c_{(1)})$, and $(p_{(2)}, c_{(2)})$ in $\mathcal{T}_{\mathcal{P}}$ (Fig. 2.3b). In a similar manner, two more vertices, $(p_{(2,1)}, c_{(2,1)})$ and $(p_{(2,1,1)}, c_{(2,1,1)})$, have been added in Fig. 2.3c. An intersection of positive area is found between cell $c_{(2,1,1)}$

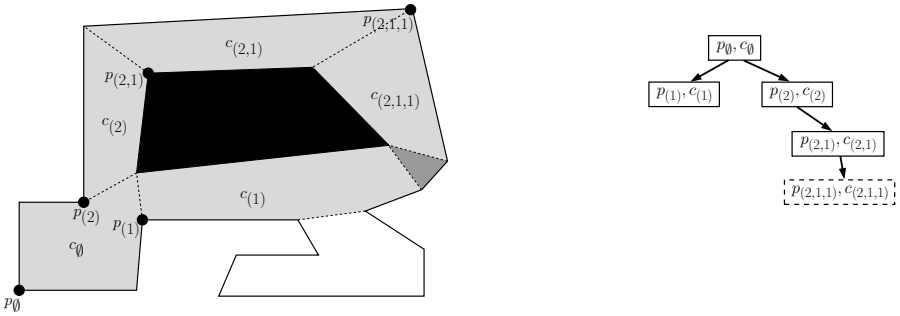
and the cell of another branch of $\mathcal{T}_{\mathcal{P}}$, namely $c_{(1)}$. To solve this *branch conflict*, the cell $c_{(2,1,1)}$ is discarded and a special marker called a *phantom wall* (thick dashed line in Fig. 2.3d) is placed where its parent gap edge was. A phantom wall serves to indicate that no branch of $\mathcal{T}_{\mathcal{P}}$ should cross a particular gap edge. The vertex $(p_{(1,2)}, c_{(1,2)})$ added in Fig. 2.3e thus can have no children. Finally, Fig. 2.3f shows the remaining vertices $(p_{(1,1)}, c_{(1,1)})$ and $(p_{(1,1,1)}, c_{(1,1,1)})$ added to $\mathcal{T}_{\mathcal{P}}$ so that the entire environment is covered and the algorithm terminates.




(a)

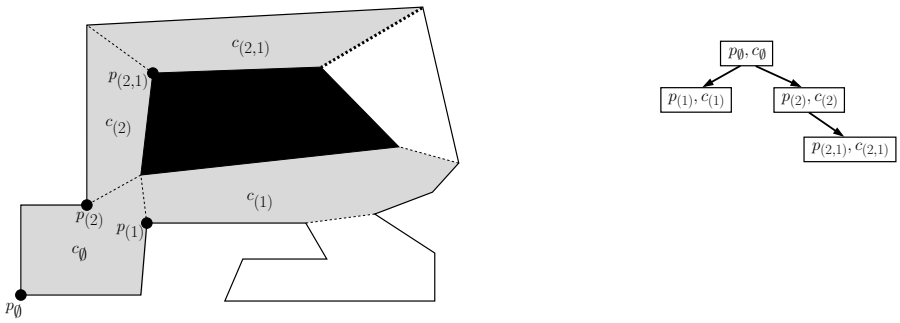



(b)

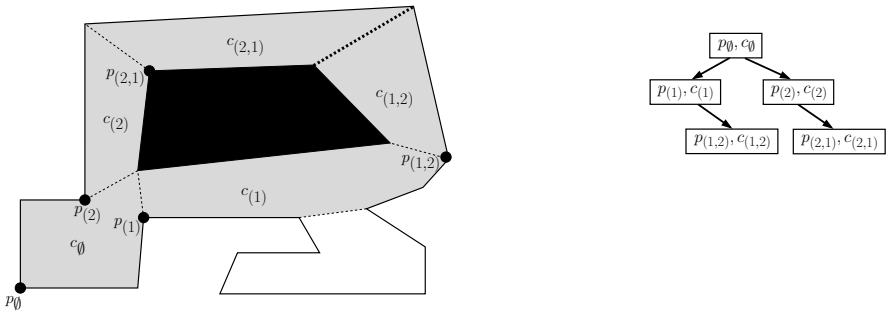



(c)

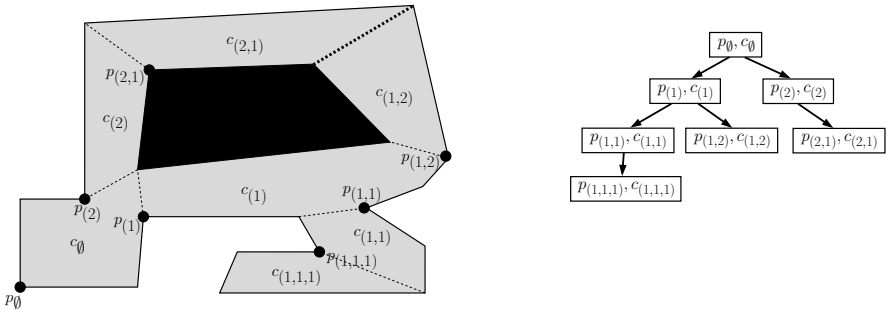
Figure 2.3. This simple example shows how the incremental partition algorithm of Table 2.1 progresses (a)-(f). Cell vantage points are shown by black disks. The portion of the environment $\mathcal{E}$ covered at each stage is shown in grey (left) along with a corresponding abstract depiction of the partition tree (right).




(d)




(e)




(f)

Figure 2.3. (continuation) A phantom wall (thick dashed line), shown first in (d), comes about when there is a *branch conflict*, i.e., when cells from different branches of the partition tree $\mathcal{T}_\mathcal{P}$ are not openly disjoint. The final partition can be used to triangulate the environment as shown in Fig. 2.4.

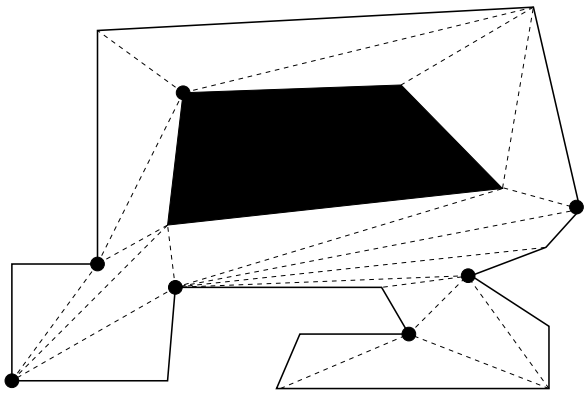


Figure 2.4. The partition tree produced by the centralized incremental partition algorithm of Table 2.1 or the distributed deployment algorithm of Table 2.6 can be used to triangulate an environment, as shown here for the simple example of Fig. 2.3. The triangulation is constructed by drawing diagonals (dashed lines) from each vantage point (black disks) to the visible environment vertices in its cell.

Now we turn our attention to the pseudocode Table 2.1 for a precise description of the algorithm. The input is the environment $\mathcal{E}$ and a single point $p_\emptyset \in V_\mathcal{E}$. The output is the partition tree $\mathcal{T}_\mathcal{P}$. We have seen that each vertex of the partition tree is a vantage point - cell pair. In particular, a cell is a data structure which stores not only a polygonal boundary, but also a label on each of the polygon's gap edges. A gap edge label takes one of four possible values: `parent`, `child`, `unexplored`, or `phantom wall`. These labels allow the following exact definition of the partition tree.

**Definition 2.1** (Partition Tree $\mathcal{T}_\mathcal{P}$)**.** *The directed rooted partition tree $\mathcal{T}_\mathcal{P}$ has*

*(i) vertex set consisting of vantage point - cell pairs produced by the incremental partition algorithm of Table 2.1, and*

Table 2.1. Centralized Incremental Partition Algorithm

INCREMENTAL_PARTITION($\mathcal{E}, p_\emptyset$)
{Compute and Insert Root Vertex into $\mathcal{T}_\mathcal{P}$}
1: $c_\emptyset \leftarrow \tilde{\mathcal{V}}(p_\emptyset)$;
2: **for** each gap edge $g$ of $c_\xi$ **do**
3: label $g$ as `unexplored` in $c_\emptyset$;
4: insert $(p_\emptyset, c_\emptyset)$ into $\mathcal{T}_\mathcal{P}$;
{Main Loop}
5: **while** any cell in $\mathcal{T}_\mathcal{P}$ has `unexplored` gap edges **do**
6: $c_\zeta \leftarrow$ any cell in $\mathcal{T}_\mathcal{P}$ with `unexplored` gap edges;
7: $g \leftarrow$ any `unexplored` gap edge of $c_\zeta$;
8: $(p_\xi, c_\xi) \leftarrow$ CHILD($\mathcal{E}, \mathcal{T}_\mathcal{P}, \zeta, g$); {See Tab. 2.2}
9: {Check for Branch Conflicts}
10: **if** there exists any cell $c_{\xi'}$ in $\mathcal{T}_\mathcal{P}$ which is in *branch conflict* with $c_\xi$ **then**
11: discard $(p_\xi, c_\xi)$;
12: label $g$ as `phantom_wall` in $c_\zeta$;
13: **else**
14: insert $(p_\xi, c_\xi)$ into $\mathcal{T}_\mathcal{P}$;
15: label $g$ as `child` in $c_\zeta$;
16: **return** $\mathcal{T}_\mathcal{P}$;

Table 2.2. Incremental Partition Subroutine

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

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

*(ii) a directed edge from vertex* $(p_\zeta, c_\zeta)$ *to vertex* $(p_\xi, c_\xi)$ *if and only if* $c_\zeta$ *has a* `child` *gap edge which coincides with a* `parent` *gap edge of* $c_\xi$.

Stepping through the pseudocode Table 2.1, lines 1-4 compute and insert the root vertex $(p_\emptyset, c_\emptyset)$ into $\mathcal{T}_\mathcal{P}$. Upon entering the main loop at line 5, line 6 selects a cell $c_\zeta$ arbitrarily from the set of cells in $\mathcal{T}_\mathcal{P}$ which have `unexplored` gap edges. Line 7 selects an arbitrary `unexplored` gap edge $g$ of $c_\zeta$. The next vantage point candidate will be placed on an endpoint of $g$ by a call on line 8 to the CHILD function of Table 2.2. The PTVUID $\xi$ is computed by the successor function on line 1 of Table 2.2. For any $d$-tuple $\zeta$ and positive integer $i$, successor$(\zeta, i)$ is simply the $(d+1)$-tuple which is the concatenation of $\zeta$ and $i$, e.g., successor$((2,1),1)) = (2,1,1)$. The CHILD function constructs a candidate vantage point $p_\xi$ and cell $c_\xi$ as follows. In the typical case, when the parent cell $c_\zeta$ has more than three edges, $c_\zeta$'s vertices are enumerated counterclockwise from $p_\zeta$, e.g., as $c_\emptyset$'s vertices in Fig. 2.3a or Fig. 2.6. In the special case of $c_\zeta$ being a triangle, e.g., as the triangular cells in Fig. 2.6, $c_\zeta$'s vertices are enumerated such that the 3 lands on $c_\zeta$'s parent gap edge. The vertex of $g$ which is odd in the enumeration is selected as $p_\xi$. Occasionally there may be *double vantage points* (colocated), e.g., as $p_{(2)}$ and $p_{(3)}$ in Fig. 2.6. We will see in Sec. 2.5.1 that this *parity-based vantage point selection scheme* is important for obtaining a special subset of the vantage points called the *sparse vantage point set*. Returning to Table 2.1, the final portion of the main loop, lines 9-14, checks whether $c_\xi$ is in *branch conflict* or $(p_\xi, c_\xi)$ should be added permanently to $\mathcal{T}_\mathcal{P}$. A cell $c_\xi$ is in branch conflict with another cell $c_{\xi'}$ if and only if $c_\xi$ and $c_{\xi'}$ are not openly disjoint (see Fig. 2.5). The main algorithm terminates when there are no more unexplored gap edges in $\mathcal{T}_\mathcal{P}$.

An important difference between our incremental partition algorithm and that

of Ganguli et al [24] is that the set of cells computed by our incremental partition is not unique. This is because the freedom in choosing cell $c_\zeta$ and gap $g$ on lines 6-7 of Table 2.1 allows different executions of the algorithm to fill the same part of the environment with different branches of $\mathcal{T}_\mathcal{P}$. This may result in different sets of phantom walls as well. A phantom wall is only created on line 11 of Table 2.1 when there is a branch conflict. This discarding may seem computationally wasteful because the environment could just be made simply connected by choosing $h$ phantom walls (one for each hole) prior to executing the algorithm. Such an approach, however, would not be amenable to distributed emulation without a priori knowledge of the environment.

The following important properties we prove for the incremental partition algorithm are similar to properties we obtain for the distributed deployment algorithm in Sec. 2.6.

**Lemma 2.1** (Star-Convexity of Partition Cells)**.** *Any partition tree vertex $(p_\xi, c_\xi)$ constructed by the incremental partition algorithm of Table 2.1, has the properties that*

*(i) the cell $c_\xi$ is star-convex, and*

*(ii) the vantage point $p_\xi$ is in the kernel of $c_\xi$.*

*Proof.* Given a star-convex set, say $S$, let $K$ be the kernel of $S$. Suppose that we obtain a new set $S'$ by truncating $S$ at a single line segment $l$ who's endpoints lie on the boundary $\partial S$. It is easy so see that the kernel of $S'$ contains $K \cap S'$, thus $S'$ must be star-convex if $K \cap S'$ is nonempty. Indeed $l$ could not possibly block line of sight from any point in $K \cap S'$ to any point $p$ in $S'$, otherwise $p$ would have been truncated. Inductively, we can obtain a set $S'$ by truncating the set $S$ at any finite number of line segments and the kernel of $S'$ will be a superset of $S' \cap K$. Now consider a partition tree vertex $(p_\xi, c_\xi)$. By definition, the visibility

polygon $\mathcal{V}(p_\xi)$ is star-convex and $p_\xi$ is in the kernel. By the above reasoning, the vertex-limited visibility polygon $\tilde{\mathcal{V}}(p_\xi)$ is also star-convex and has $p_\xi$ in its kernel because $\tilde{\mathcal{V}}(p_\xi)$ can be obtained from $\mathcal{V}(p_\xi)$ by a finite number of line segment truncations (lines 8 and 9 of Table 2.2). Likewise, $c_\xi$ must be star-convex with $p_\xi$ in its kernel because $c_\xi$ is obtained from $\tilde{\mathcal{V}}(p_\xi)$ by a finite number of line segment truncations at the parent gap edge and phantom walls. □

**Theorem 2.1** (Properties of the Incremental Partition Algorithm)**.** *Suppose the incremental partition algorithm of Table 2.1 is executed on an environment $\mathcal{E}$ with $n$ vertices and $h$ holes. Then*

*(i) the algorithm returns in finite time a partition tree $\mathcal{T}_\mathcal{P}$ such that every point in the environment is visible to some vantage point,*

*(ii) the visibility graph of the vantage points $\mathcal{G}_{\text{vis}}\mathcal{E}(\{p_\xi | (p_\xi, c_\xi) \in \mathcal{T}_\mathcal{P}\})$ consists of a single connected component,*

*(iii) the final number of vertices in $\mathcal{T}_\mathcal{P}$ (and thus the total number of vantage points) is no greater than $n + 2h - 2$,*

*(iv) there exist environments where the final number of vertices in $\mathcal{T}_\mathcal{P}$ is equal to the upper bound $n + 2h - 2$, and*

*(v) the final number of phantom walls is precisely $h$.*

*Proof.* We prove the statements in order. The algorithm processes `unexplored` gap edges one by one and terminates when there are no more `unexplored` gap edges. Once an `unexplored` gap edge has been processed, it is never processed again because its label changes to `phantom_ wall` or `child`. Gap edges of cells are diagonals of the environment and there are no more than $\binom{n}{2} = \frac{n^2-n}{2}$ possible diagonals, which is finite, therefore the algorithm must terminate in finite time. Lemma 2.1 guarantees that if the entire environment is covered by cells of $\mathcal{T}_\mathcal{P}$, then every point is visible to some vantage point. Suppose the final set of cells does not cover the entire environment. Then there must be a portion of the environment which is topologically isolated from the rest of the environment by

phantom walls, otherwise an `unexplored` gap edge would have expanded into that region. However, this would mean that a phantom wall was created at the `parent` gap edge of a candidate cell which was not in branch conflict. This is not possible because a phantom wall is only ever created if there is a branch conflict (lines 9-10 Table 2.1). This completes the proof of statement (i).

Statement (ii) follows from Lemma 2.1 together with the fact that every vantage point is placed on the boundary of its parent's cell. Given two vantage points in $\mathcal{T}_{\mathcal{P}}$, say $p_\xi$ and $p_{\xi'}$, a path through $\mathcal{G}_{\text{vis}}\mathcal{E}(\{p_\xi|(p_\xi, c_\xi) \in \mathcal{T}_{\mathcal{P}}\})$ from $p_\xi$ to $p_{\xi'}$ can be constructed as follows. Follow parent-child visibility links up to the root vantage point $p_\emptyset$, then follow parent-child visibility links from $p_\emptyset$ down to $p_{\xi'}$. Since such a path can always be constructed between any pair of vantage points, $\mathcal{G}_{\text{vis}}\mathcal{E}(\{p_\xi|(p_\xi, c_\xi) \in \mathcal{T}_{\mathcal{P}}\})$ must consist of a single connected component.

For statement (iii), we triangulate $\mathcal{E}$ by triangulating the cells of $\mathcal{T}_{\mathcal{P}}$ individually as in Fig. 2.4. Each cell $c_\xi$ is triangulated by drawing diagonals from $p_\xi$ to the vertices of $c_\xi$. The total number of triangles in any triangulation of a polygonal environment with holes is $n+2h-2$ (Lemma 5.2 in [7]). Since there is at least one triangle per cell and at most one vantage point per cell, the number of vantage points cannot exceed the maximum number of triangles $n + 2h - 2$.

Statement (iv) is proven by the example in Fig. 2.7a.

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

□

### 2.5.1 A Sparse Vantage Point Set

Suppose we were to deploy robotic agents onto the vantage points produced by the incremental partition algorithm (one agent per vantage point). Then, as Theorem 2.1 guarantees, we would achieve our goal of complete visibility coverage with connectivity. The number of agents required would be no greater than the number of vantage points, namely $n+2h-2$. This upper bound, however, can be greatly improved upon. In order to reduce the number of vantage points agents must deploy to, the postprocessing algorithm in Table 2.3 takes the partition tree output by the incremental partition algorithm and labels a subset of the vantage points called the *sparse vantage point set*. Starting at the leaves of the partition tree and working towards the root, vantage points are labeled either `nonsparse` or `sparse` according to criterion on line 2 of Table 2.3. As proven in Theorem 2.2 below, the sparse vantage points are suitable for the coverage task and their cardinality has a much better upper bound than the full set of vantage points. All the vantage points in the example of Fig. 2.3 are sparse. Fig. 2.6 shows an example of when only a proper subset of the vantage points is sparse.

**Lemma 2.2** (Properties of a Child Vantage Point of a Triangular Cell)**.** *Let $(p_\xi, c_\xi)$ be a partition tree vertex constructed by the incremental partition algorithm of Table 2.1 and suppose $c_\xi$ has a parent cell $c_\zeta$ which is a triangle. Then $p_\xi$ is in the kernel of $p_\zeta$. Furthermore, if $p_\zeta$ has a parent vantage point $p_{\zeta'}$ (the grandparent of $p_\xi$), then $p_\xi$ is visible to $p_{\zeta'}$.*

*Proof.* The kernel of a triangular (and thus convex) cell $c_\zeta$ is all of $c_\zeta$. By Lemma 2.1, $p_{\zeta'}$ is in the kernel of $c_{\zeta'}$. According to the parity-based vantage

Table 2.3. Postprocessing of Partition Tree

LABEL_VANTAGE_POINTS($\mathcal{E}, \mathcal{T}_\mathcal{P}$)

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

point selection scheme (line 5 of Table 2.2), $p_\xi$ is located at a point common to $c_{\zeta'}$, $c_\zeta$, and $c_\xi$, therefore $p_\xi$ is in the kernel of $c_\zeta$ and visible to $c_{\zeta'}$. □

**Theorem 2.2** (Properties of the Sparse Vantage Point Set)**.** *Suppose the incremental partition algorithm of Table 2.1 is executed to completion on an environment $\mathcal{E}$ with $n$ vertices and $h$ holes and the vantage points of the resulting partition tree are labeled by the algorithm in Table 2.3. Then*

*(i) every point in the environment is visible to some sparse vantage point,*

*(ii) the visibility graph of the sparse vantage points $\mathcal{G}_{\text{vis}}\mathcal{E}(\{p_\xi | (p_\xi, c_\xi) \in \mathcal{T}_\mathcal{P}\})$ consists of a single connected component,*

*(iii) the number of points in $\mathbb{R}^2$ where at least one sparse vantage point is located is no greater than $\lfloor \frac{n+2h-1}{2} \rfloor$, and*

*(iv) there exist environments where the upper bound $\lfloor \frac{n+2h-1}{2} \rfloor$ in (iii) is met.*

*Proof.* Statements (i) and (ii) follow directly from Lemma 2.2 together with statements (i) and (ii) of Theorem 2.1.

For statement (iii) we use a triangulation argument similar to that used in [24] for environments without holes. We use the same triangulation as in the proof

of Theorem 2.1 (Fig. 2.4). The total number of triangles in any triangulation of a polygonal environment with holes is $n + 2h - 2$ (Lemma 5.2 in [7]). Suppose we can assign at least one unique triangle to $p_\emptyset$ whenever $p_\emptyset$ is sparse and at least two unique triangles to all other sparse vantage point locations. Then using the formula for the total number of triangles, we see the total number of sparse vantage point locations is upper bounded by

$$\left\lfloor \frac{(n + 2h - 2) + 1}{2} \right\rfloor = \left\lfloor \frac{n + 2h - 1}{2} \right\rfloor,$$

which is the desired result. Indeed we can make such an assignment of triangles to sparse vantage point locations. Our argument relies on the parity-based vantage point selection scheme and the criterion for labeling a vantage point as `sparse` on line 2 of Table 2.3. To any sparse vantage point location, say of $p_\xi$ other than the root, we assign one triangle in the parent cell. The triangle in the parent cell is the triangle formed by its parent gap edge together with its parent's vantage point. To each sparse vantage point location, say of $p_\xi$, including the root, we assign additionally one triangle in the cell $c_\xi$. If $c_\xi$ has no children, then any triangle in $c_\xi$ can be assigned to $p_\xi$. If $c_\xi$ has children (in which case it must have greater than onen triangle) we need to check that it has more triangles than child vantage point locations with odd parity. Suppose $c_\xi$ has an even number of edges. Then this number of edges can be written $2m$ where $m \geq 2$. The number of triangles in $c_\xi$ is $2m - 2$ and the number of odd parity vertices in $c_\xi$ where child vantage points could be placed is $m - 1$. This means at most $m - 1$ triangles in $c_\xi$ are assigned to odd parity child vantage point locations, which leaves $(2m - 2) - (m - 1) = m - 1 \geq 1$ triangles to be assigned to the location of $p_\xi$. The case of $c_\xi$ having an odd number of edges is proven analogously.

Statement (iv) is proven by the example in Fig. 2.7.

□

## 2.6 Distributed Deployment Algorithm

In this section we describe how a group of mobile robotic agents can distributedly emulate the incremental partition and vantage point labeling algorithms of Sec. 2.5, thus solving the Distributed Visibility-Based Deployment Problem with Connectivity. We first give a rough overview of the algorithm, called DEPLOY_DEPTH-FIRST(), and later address the details with aid of the pseudocode in Tables 2.6. Each agent $i$ has a local variable mode$^{[i]}$, among others, which takes a value `lead`, `proxy`, or `explore`. For short, we call an agent in `lead` mode a *leader*, an agent in `proxy` mode a *proxy*, and an agent in `explore` mode an *explorer*. Agents may switch between modes (see Fig. 2.8a) based on certain asynchronous events. Leaders settle at sparse vantage points and are responsible for maintaining in their memory a distributed representation of the partition tree $\mathcal{T}_\mathcal{P}$ consistent with Definition 2.1. By distributed representation we mean that each leader $i$ retains in its memory up to two *vertices of responsibility*, $(p_1^{[i]}, c_1^{[i]})$ and $(p_2^{[i]}, c_2^{[i]})$, and it knows which gap edges of those vertices lead to the parent and child vertices in $\mathcal{T}_\mathcal{P}$.[3] We call $(p_1^{[i]}, c_1^{[i]})$ the *primary vertex* of agent $i$ and $(p_2^{[i]}, c_2^{[i]})$ the *secondary vertex*. A leader typically has only a primary vertex in its memory and may have also a secondary only if it is either positioned (1) at a double vantage point, or (2) at a sparse vantage point adjacent to a nonsparse vantage point. Each cell in a leader's memory has a status which takes the value `retracting`, `stable`, or `fertile` (see Fig. 2.8b). Only when a cell has attained fertile status can any child $\mathcal{T}_\mathcal{P}$ vertices be added at its unexplored gap edges. Intuitively, the reason a cell must go through two statuses before becoming fertile is that two

[3]The subscripts of the *vertices of responsibility* are not to be confused with PTVUIDs, i.e., $(p_1^{[i]}, c_1^{[i]})$ and $(p_2^{[i]}, c_2^{[i]})$ are not in general the same as $(p_{(1)}, c_{(1)})$ and $(p_{(2)}, c_{(2)})$.

things had to be done to a newly constructed cell in the incremental partition algorithm: the cell had to be (1) truncated at existing phantom walls, and then (2) deleted if it was in branch conflict. The job of a proxy agent is to assist leaders in advancing the status of their cells towards `fertile` by proxying communication with other leaders (see Fig 2.5d). Any agent which is not a leader or proxy is an explorer. Explorers merely move in depth-first order systematically about $\mathcal{T}_{\mathcal{P}}$ in search of opportunity to serve as a proxy or leader (see Fig. 2.9). To simplify the presentation, let us assume for now that, as in the example Fig. 2.3, no double vantage points or triangular cells occur. Under this assumption, each leader will be responsible for only one $\mathcal{T}_{\mathcal{P}}$ vertex, its primary vertex, and all vantage points will be sparse. The deployment begins with all agents colocated at the first vantage point $p_\emptyset$. One agent, say agent 0, is initialized to `lead` mode with the first cell $c_{\xi_1}^{[0]} = c_\emptyset = \tilde{\mathcal{V}}(p_\emptyset)$ in its memory. All other agents are initialized to `explore` mode. Agent 0 can immediately advance the status of $c_\emptyset$ to `fertile` because it cannot possibly be in branch conflict (no other cells even exist yet); in general, however, cells can only transition between statuses when a proxy tour is executed. Agent 0 sees all the explorers in its cell and assigns as many as necessary to become leaders so that there will be one new leader positioned on each unexplored gap edge of $c_\emptyset$. The new leader agents move concurrently to their new respective vantage points while all remaining explorer agents move towards the next cell in their depth-first ordering. When a leader first arrives at its vantage point, say $p_\xi$, of the cell $c_\xi$, it initializes $c_\xi$ to have status `retracting` and boundary equal to the portion of $\tilde{\mathcal{V}}(p_\xi)$ which is across the parent gap edge and extends away from the parent's cell. When an explorer agent comes to such a newly created retracting cell, the leader assigns that explorer to become a proxy and follow a proxy tour which traverses all the gap edges of $c_\xi$. During the proxy tour, the proxy agent is

able to communication with any leader of a `fertile` cell that might be in branch conflict with the $c_\xi$. The cell $c_\xi$ is thus truncated as necessary to ensure it is not in branch conflict with any `fertile` cell. When this first proxy tour is complete, the status of $c_\xi$ is advanced to `stable`. The leader of $c_\xi$ then assigns a second proxy tour which again traverses all the gap edges of $c_\xi$. During this second proxy tour, the leader communicates, via proxy, with all leaders of stable cells which come into line of sight of the proxy. If a branch conflict is detected between $c_\xi$ and another stable cell, the agents have a *standoff*: they agree to delete the $\mathcal{T}_\mathcal{P}$ vertex with PTVUID which is larger according to the following total ordering.

**Definition 2.2** (PTVUID Total Ordering)**.** *Let $\xi_1$ and $\xi_2$ be distinct PTVUIDs. If $\xi_1$ and $\xi_2$ do not have equal depth, then $\xi_1 < \xi_2$ if and only if the depth of $\xi_1$ is less than the depth of $\xi_2$. If $\xi_1$ and $\xi_2$ do have equal depth, then $\xi_1 < \xi_2$ if and only if $\xi_1$ is lexicographically smaller than $\xi_2$.*[4]

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

We now turn our attention to pseudocode Table 2.6 to describe

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

DEPLOY_DEPTH-FIRST() more precisely. The algorithm consists of three threads which run concurrently in each agent: communication (lines 1-5), navigation (lines 6-11), and internal state transition (lines 12-23). An outline of the local variables used for these threads is shown in Table 2.4 and 2.5. The communication thread tracks the internal states of all an agent's visibility neighbors. One could design a custom communication protocol for the deployment which would make more efficient use of communication bandwidth, however, we find it simplifies the presentation to assume agents have direct access to their visibility neighbors' internal states via the data structure Neighbor_Data$^{[i]}$. The navigation thread has the agent follow, at maximum velocity $u_{\max}$, a queue of waypoints called Route$^{[i]}$ as long as the internal state component $c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set is empty (it is only ever nonempty for a proxy agent and its meaning is discussed further in Section 2.6.2). The waypoints can be represented in a local coordinate system established by the agent every time it enters a new cell, e.g., a polar coordinate system with origin at the cell's vantage point. In the internal state transition thread, an agent switches between `lead`, `proxy`, and `explore` modes. The agent reacts to different asynchronous events depending on what mode it is in. We treat the details of the different mode behaviors and corresponding subroutines in the following Sections 2.6.1, 2.6.2, and 2.6.3.

### 2.6.1 Leader Behavior

The `lead` portion of the internal state transition thread (lines 13-16 of Table 2.6) consists of three subroutines: ATTEMPT_CELL_CONSTRUCTION(), LEAD(), and PROPAGATE_SPARSE_VANTAGE_POINT_INFORMATION(). In ATTEMPT_CELL_CONSTRUCTION() (Table 2.7), the leader agent attempts to

Table 2.4. Agent Local Variables for Distributed Deployment

| Use | Name | Brief Description |
|---|---|---|
| Communication | $\text{UID}^{[i]} := i$ | agent Unique IDentifier |
| | $\text{In\_Buffer}^{[i]}$ | FIFO queue of messages received from other agents |
| | $\text{Neighbor\_Data}^{[i]}$ | data structure which tracks relevant state information of visibility neighbors |
| | $\text{state\_change\_interrupt}^{[i]}$ | boolean, `true` if and only if internal state has changed between the last and current iteration of the communication thread |
| | $\text{new\_visible\_agent\_interrupt}^{[i]}$ | boolean, `true` if and only if a new agent became visible between the last and current iteration of the communication thread |
| Navigation | $\text{Route}^{[i]}$ | FIFO queue of waypoints |
| | $p^{[i]}$, $\dot{p}^{[i]}$, $u$ | position, velocity, and velocity input |
| Internal State | $\text{mode}^{[i]}$ | agent mode takes a value `lead`, `proxy`, or `explore` |
| | $\text{Vantage\_Points}^{[i]} := (p^{[i]}_{\xi_1}, p^{[i]}_{\xi_2})$ | vantage points used in `lead` mode for distributed representation of $\mathcal{T}_\mathcal{P}$; may have size 0, 1, or 2; each $p_\xi$ may be labeled either `sparse` or `nonsparse` |
| | $\text{Cells}^{[i]} := (c^{[i]}_{\xi_1}, c^{[i]}_{\xi_2})$ | cells used in `lead` mode for distributed representation of $\mathcal{T}_\mathcal{P}$; may have size 0, 1, or 2; cell fields shown in Tab. 2.5 |
| | $\text{Parent\_Gap\_Edges}^{[i]} := (g^{[i]}_{\xi_1}, g^{[i]}_{\xi_2})$ | parent gap edges used in `lead` mode for initializing cells in $\text{Cells}^{[i]}$; may have size 0, 1, or 2 |
| | $c^{[i]}_{\xi_{\text{proxied}}}$ | used in `proxy` mode as local copy of cell being proxied |
| | $\xi^{[i]}_{\text{current}}$, $\xi^{[i]}_{\text{last}}$ | PTVUIDs of current and last $\mathcal{T}_\mathcal{P}$ vertices visited in depth-first search; used in `explore` mode to navigate $\mathcal{T}_\mathcal{P}$ |

Table 2.5. Cell Data Fields for Distributed Deployment

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

construct a cell in such a way that the cell is guaranteed to contain at least one triangle which is not in any other cell of the distributed $\mathcal{T}_\mathcal{P}$ representation. Having at least one unique triangle is important for proving an upper bound on number of required agents in Theorem 2.3. To this end, the leader may switch to proxy mode and proxy for another leader in line of sight. If, even after proxying, the agent cannot guarantee a unique triangle, then it places a phantom wall at the parent gap edge of the cell and deletes the cell. If the agent can guarantee a unique triangle, then it initializes the cell to `retracting` status and waits for a proxy agent to help it advance the cell's status towards `fertile`.

In LEAD() (Table 2.8), the agent already has initialized cell(s) in its memory. Being reponsible for cells means that the leader agent may have to assign itself a secondary $\mathcal{T}_\mathcal{P}$ vertex which is the child of its primary vertex, assign another

Table 2.6. Distributed Deployment Algorithm

DEPLOY_DEPTH-FIRST()

{ Communication Thread }

**while** `true` **do**
  in_message ← In_Buffer$^{[i]}$.PopFirst();
  update Neighbor_Data$^{[i]}$ according to in_message;
  **if** state_change_interrupt$^{[i]}$ **or** visible_agent_interrupt$^{[i]}$ **then**
    broadcast internal state information;

{ Navigation Thread }

**while** `true` **do**
  **while** Route$^{[i]}$ is not empty **and** $p^{[i]} \neq$ Route$^{[i]}$.First() **and** $c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set is empty **do**
    $u^{[i]} \leftarrow$ velocity with magnitude $u_{\max}$ and direction towards Route$^{[i]}$.First();
  $u^{[i]} \leftarrow 0$;
  **if** $p^{[i]} ==$ Route$^{[i]}$.First() **then**
    Route$^{[i]}$.PopFirst();

{ Internal State Transition Thread }

**while** `true` **do**
  **if** mode$^{[i]} ==$ `lead` **then**
    ATTEMPT_CELL_CONSTRUCTION(); { See Tab. 2.7 }
    LEAD(); { See Tab. 2.8 }
    PROPAGATE_SPARSE_VANTAGE_POINT_INFORMATION(); { See Tab. 2.9 }
  **else if** mode$^{[i]} ==$ `proxy` **then**
    **if** $c_{\text{proxied}}$.status == `retracting` **then**
      PROXY_RETRACTING_CELL(); { See Tab. 2.10 }
    **else if** $c_{\text{proxied}}$.status == `stable` **then**
      PROXY_STABLE_CELL(); { See Tab. 2.11 }
  **else if** mode$^{[i]} ==$ `explore` **then**
    EXPLORE(); { See Tab. 2.12 }

Table 2.7. Distributed Deployment Subroutine

ATTEMPT_CELL_CONSTRUCTION()

1: **if** there is a vantage point $p_\xi$ in Vantage_Points[$i$] for which no cell in Cells$^{[i]}$ has yet been constructed
   **and** $p^{[i]} == p_\xi$ **then**
2:  **if** Neighbor_Data$^{[i]}$ contains a cell $c_{\xi'}$ such that $c_{\xi'}$.proxy_uid $== i$ **then**
3:  { Proxy for another leader }
4:  mode$^{[i]} \leftarrow$ `proxy`; Route$^{[i]} \leftarrow$ tour which traverses all gap edges of $c'_\xi$ and returns to $p_\xi$;
5:  **else if** Neighbor_Data$^{[i]}$ shows any stable or fertile cell $c_{\xi'}$ having a gap edge coinciding with $g_\xi$
   **and** $\xi'$ is not the parent PTVUID of $\xi$ **then**
6:  { Label phantom wall if not at least one unique triangle }
7:  **if** Cells$^{[i]}$ is empty **then**
8:  label $g^{[i]}_{\xi_1}$ as `phantom_wall`;
9:  clear $g^{[i]}_{\xi_1}$ and $p^{[i]}_{\xi_1}$;
10:  mode$^{[i]} \leftarrow$ `explore`; swap $\xi^{[i]}_{\text{last}}$ and $\xi^{[i]}_{\text{current}}$;
11:  **else if** Cells$^{[i]}$ contains exactly one cell **then**
12:  label any gap edges of $c^{[i]}_{\xi_1}$ that coincide with $g^{[i]}_{\xi_2}$ as `phantom_wall`;
13:  clear $g^{[i]}_{\xi_2}$ and $p^{[i]}_{\xi_2}$; Route$^{[i]} \leftarrow$ straight path to $p^{[i]}_{\xi_1}$;
14:  **else if** Neighbor_Data$^{[i]}$ shows no other agent about to construct a cell $c_{\xi'}$ where $\xi' < \xi$ **then**
15:  { Compute initial cell }
16:  $c_\xi \leftarrow \tilde{\mathcal{V}}(p_\xi)$;
17:  truncate $c_\xi$ at $g_\xi$ such that only the portion remains which is across $g$ from the parent;
18:  **for** each gap edge $g'$ of $c_\xi$ **do**
19:  **if** $g' == g$ **then**
20:  label $g'$ as `parent` in $c_\xi$;
21:  **else**
22:  label $g'$ as `unexplored` in $c_\xi$;
23:  insert $c_\xi$ into Cells$^{[i]}$;

Table 2.8. Distributed Deployment Subroutine

LEAD()

1: **if** Cells$^{[i]}$ contains only a single fertile cell $c_{\xi_1}^{[i]}$
**and** $c_{\xi_1}^{[i]}$ is triangle with one unexplored gap edge $g_\xi$
**and** $g_\xi$ has not been assigned a leader **then**
2: { Assign self a secondary vertex at child of primary vertex }
3: $p_{\xi_2}^{[i]} \leftarrow p_\xi$; $g_{\xi_2}^{[i]} \leftarrow g_\xi$;
4: Route$^{[i]} \leftarrow$ straight line path to $p_\xi$;
5: label $g_\xi$ on $c_{\xi_1}^{[i]}$ as `child` and as having leader $i$;
6: **else if** Cells$^{[i]}$ contains cell $c_\zeta$ with double child vantage point $p_\xi = p_{\xi'}$ where $\xi < \xi'$
**and** Neighbor_Data$^{[i]}$ contains an agent $j$ with $c_\xi$ in Cells$^{[j]}$
**and** $p_\xi$ is labeled `sparse`
**and** gap edge $g_{\xi'}$ is unexplored **then**
7: { Assign other leader a secondary vertex at double vantage point }
8: label $g_{\xi'}$ on $c_\zeta$ as `child` and having leader $j$;
9: **else if** Neighbor_Data$^{[i]}$ shows explorer agent $j$ such that $c_\xi = c_{\xi^{[j]}_{\text{current}}}$ is fertile in Cells$^{[i]}$ **then**
10: $\xi' \leftarrow$ PTVUID of next vertex in *depth-first ordering*;
11: **if** there is an unexplored gap edge $g_{\xi'}$ of $c_\xi$
**and** ( $p_{\xi'}$ is single vantage point
**or** double vantage point with colocated vantage point `nonsparse` in Neighbor_Data$^{[i]}$ ) **then**
12: { Assign explorer to become leader of child vertex }
13: label $g_{\xi'}$ in $c_\xi$ as `child` and having leader $j$;
14: **if** Neighbor_Data$^{[i]}$ contains an explorer agent $j$
**and** Cells$^{[i]}$ contains a cell $c_\xi = c_{\xi^{[j]}_{\text{current}}}$ with $c_\xi$.status $\neq$ `fertile`
**and** $c_\xi$.proxy_uid $== \emptyset$ **then**
15: { Assign explorer as proxy }
16: $c_\xi$.proxy_uid $\leftarrow j$;
17: **else if** Neighbor_Data$^{[i]}$ contains a leader agent $j$ with Cells$^{[j]}$ empty
**and** Cells$^{[i]}$ contains a retracting cell $c_\xi$ **and** $c_\xi$.proxy_uid $== \emptyset$ **then**
18: { Assign leader as proxy }
19: $c_\xi$.proxy_uid $\leftarrow j$;
20: **if** Neighbor_Data$^{[i]}$ contains a child gap edge $g_\xi$ with agent $i$ labeled as its leader
**and** $p_\xi$ is not in Vantage_Points$^{[i]}$ **then**
21: { Accept leadership of second cell at double vantage point }
22: $p_{\xi_2}^{[i]} \leftarrow p_\xi$; $g_{\xi_2}^{[i]} \leftarrow g_\xi$;

(continued)

(continuation)

{ Respond to proxy events }
**if** cell $c_\xi$ in Cells$^{[i]}$ corresponds to cell $c^{[j]}_{\xi_{\text{proxied}}}$ in Neighbor_Data$^{[i]}$ **then**
  **if** $c^{[j]}_{\xi_{\text{proxied}}}$ has been truncated at a fertile cell **then**
    perform the same truncation on $c_\xi$;
  **if** $c_\xi$.Wait_Set $\neq c^{[j]}_{\xi_{\text{proxied}}}$.Wait_Set **then**
    $c_\xi$.Wait_Set $\leftarrow c^{[j]}_{\xi_{\text{proxied}}}$.Wait_Set;
  **if** agent $j$ has completed the proxy tour **then**
    **if** $c_\xi$ has a phantom wall at its parent gap edge **then**
      **if** Cells$^{[i]}$ contains exactly one cell **then**
        clear $p^{[i]}_{\xi_1}$, $g^{[i]}_{\xi_1}$, and $c^{[i]}_{\xi_1}$;
        mode$^{[i]} \leftarrow$ `explore`;
      **else if** Cells$^{[i]}$ contains two cells **then**
        clear $p^{[i]}_{\xi_2}$, $g^{[i]}_{\xi_2}$, and $c^{[i]}_{\xi_2}$;
        Route$^{[i]} \leftarrow$ straight path to $p^{[i]}_{\xi_1}$;
    **else**
      advance $c_\xi$.status;
      $c_\xi$.proxy_uid $\leftarrow \emptyset$;
{ Label phantom walls }
**if** Neighbor_Data$^{[i]}$ shows a stable cell $c^{[j]}_{\xi_{\text{proxied}}}$ in *branch conflict* with $c_\xi$ in Cells$^{[i]}$
  **and** $\xi^{[j]}_{\text{proxied}} < \xi$ **then**
  { Yield to competing stable cell }
  label parent gap edge of $c_\xi$ as `phantom_wall`;
**if** Neighbor_Data$^{[i]}$ shows a phantom wall coinciding with gap edge $g$ of cell $c_\xi$ in Cells$^{[i]}$ **then**
  label $g$ as `phantom_wall` in $c_\xi$;
**if** Neighbor_Data$^{[i]}$ shows stable cell $c^{[j]}_{\xi_{\text{proxied}}}$ in *branch conflict* with stable cell $c_\xi$ in Cells$^{[i]}$
  **and** $\xi^{[j]}_{\text{proxied}} < \xi$ **then**
  label parent gap edge of $c_\xi$ as `phantom_wall`;

Table 2.9. Distributed Deployment Subroutine

PROPAGATE_SPARSE_VANTAGE_POINT_INFORMATION()

1: { Label a vantage point in Vantage_Points$^{[i]}$ as `sparse` or `nonsparse` }
2: **if** there is an unlabeled vantage point $p_\xi$ in Vantage_Points$^{[i]}$ with fertile cell $c_\xi$ in Cells$^{[i]}$
**and** ( $(p_\xi, c_\xi)$ is a leaf **or** Cells$^{[i]}$ and Neighbor_Data$^{[i]}$ show all child vantage points have been labeled ) **then**
3: **if** $|V_{c_\xi}| == 3$ **and** Cells$^{[i]}$ or Neighbor_Data$^{[i]}$ shows a child vantage point labeled `sparse` **then**
4: label $p_\xi$ as `nonsparse`;
5: **else**
6: label $p_\xi$ as `sparse`;
7: { Acquire a nonsparse vertex from an agent higher in the partition tree}
8: **if** Cells$^{[i]}$ contains exactly one cell $c_\xi$ with $p_\xi$ labeled `sparse` **and** $p^{[i]} == p_\xi$
**and** Neighbor_Data$^{[i]}$ shows a cell $c_\zeta$ which is the parent of $c_\xi$ **and** $p_\zeta$ is labeled `nonsparse` **then**
9: insert $c_\zeta$ into Cells$^{[i]}$ and $p_\zeta$ into Vantage_Points$^{[i]}$;
10: { Give up a nonsparse vertex to an agent lower in the partition tree }
11: **if** Neighbor_Data$^{[i]}$ shows a leader agent $j$ with $p^{[j]}_{\xi_1}$ labeled `sparse` **and** $c^{[i]}_{\xi_2} == c^{[j]}_{\xi_2}$ **and** $\xi^{[j]}_2$ is the parent PTVUID of $\xi^{[i]}_1$ **then**
12: clear $p^{[i]}_{\xi_2}$, $g^{[i]}_{\xi_2}$, and $c^{[i]}_{\xi_2}$; Route$^{[i]}$ $\leftarrow$ straight path to $p^{[i]}_{\xi_1}$;

Table 2.10. Distributed Deployment Subroutine

PROXY_RETRACTING_CELL()

1: **if** Route$^{[i]}$ is nonempty **then**
2: **if** Neighbor_Data$^{[i]}$ shows fertile cell $c_\xi$ in *branch conflict* with $c^{[i]}_{\xi_{\text{proxied}}}$ **then**
3: { Truncate $c_{\xi_{\text{proxied}}}$ at fertile cell }
4: truncate $c^{[i]}_{\xi_{\text{proxied}}}$ at $c_\xi$;
5: **if** Neighbor_Data$^{[i]}$ shows stable cell $c_\xi$ in *branch conflict* with $c^{[i]}_{\xi_{\text{proxied}}}$ **and** $c_\xi$.proxy_uid $\neq \emptyset$ **and** $\xi^{[i]}_{\text{proxied}}$ is not in $c_\xi$.Wait_Set **then**
6: { Add competing cell to $c_{\xi_{\text{proxied}}}$.Wait_Set }
7: insert $\xi$ into $c_{\xi_{\text{proxied}}}$.Wait_Set;
8: **if** Neighbor_Data$^{[i]}$ shows $c_\xi$ no longer in *branch conflict* with $c^{[i]}_{\xi_{\text{proxied}}}$ for some $\xi$ in $c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set
**or** ( Neighbor_Data$^{[i]}$ shows fertile cell $c_\xi$ such that $c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set contains $\xi$ )
**or** ( Neighbor_Data$^{[i]}$ shows stable cell $c_\xi$ for some $\xi$ in $c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set
**and** $\xi > \xi^{[i]}_{\text{proxied}}$ **and** $c_\xi$.Wait_Set contains $\xi^{[i]}_{\text{proxied}}$ ) **then**
9: { Remove competing cell from $c_{\xi_{\text{proxied}}}$.Wait_Set }
10: remove $\xi$ from $c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set;
11: **else if** Route$^{[i]}$ is empty **then**
12: { End tour and enter previous mode, explore or lead }
13: **if** Vantage_Points$^{[i]}$ is empty **then**
14: mode$^{[i]} \leftarrow$ `explore`;
15: **else**
16: mode$^{[i]} \leftarrow$ `lead`;
17: clear $c^{[i]}_{\xi_{\text{proxied}}}$;

Table 2.11. Distributed Deployment Subroutine

PROXY_STABLE_CELL()

1: **if** Route$^{[i]}$ is nonempty **and** the parent gag edge of $c^{[i]}_{\xi_{\text{proxied}}}$ is not phantom wall **then**
2: **if** ( Neighbor_Data$^{[i]}$ shows stable cell $c_\xi$ in *branch conflict* with $c^{[i]}_{\xi_{\text{proxied}}}$ **and** $\xi < \xi^{[i]}_{\text{proxied}}$ ) **or** Neighbor_Data$^{[i]}$ shows a phantom wall coinciding with parent gap edge of $c^{[i]}_{\xi_{\text{proxied}}}$ **then**
3: { Yield to competing stable cell }
4: label parent gap edge of $c^{[i]}_{\xi_{\text{proxied}}}$ as `phantom_wall`; clear $c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set;
5: Route$^{[i]} \leftarrow$ straight path towards midpoint of $g_{\xi_{\text{current}}}$;
6: **if** Neighbor_Data$^{[i]}$ shows retracting cell $c_\xi$ in *branch conflict* with $c^{[i]}_{\xi_{\text{proxied}}}$ **and** $c_\xi$.proxy_uid $\neq \emptyset$ **and** $c_\xi$.Wait_Set does not contain $\xi^{[i]}_{\text{proxied}}$ **then**
7: { Add competing cell to $c_{\xi_{\text{proxied}}}$.Wait_Set }
8: insert $\xi$ into $c_{\xi_{\text{proxied}}}$.Wait_Set;
9: **if** Neighbor_Data$^{[i]}$ shows $c_\xi$ no longer in *branch conflict* with $c^{[i]}_{\xi_{\text{proxied}}}$ for some $\xi$ in $c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set **or** ( Neighbor_Data$^{[i]}$ contains stable cell $c_\xi$ where $c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set contains $\xi$ ) **or** ( Neighbor_Data$^{[i]}$ shows retracting cell $c_\xi$ for some $\xi$ in $c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set **and** $\xi > \xi^{[i]}_{\text{proxied}}$ **and** $c_\xi$.Wait_Set contains $\xi^{[i]}_{\text{proxied}}$ ) **then**
10: { Remove competing cell from $c_{\xi_{\text{proxied}}}$.Wait_Set }
11: remove $\xi$ from $c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set;
12: **else if** Route$^{[i]}$ is empty **then**
13: { End tour and become explorer }
14: mode$^{[i]} \leftarrow$ `explore`;
15: **if** parent gap edge of $c^{[i]}_{\xi_{\text{proxied}}}$ is phantom wall **then**
16: swap $\xi_{\text{last}}$ and $\xi_{\text{current}}$;
17: clear $c_{\xi_{\text{proxied}}}$;

Table 2.12. Distributed Deployment Subroutine

EXPLORE()

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

leader agent a secondary vertex at a double vantage point, assign an explorer agent to become a leader of a child vertex, assign another agent to proxy, or accept leadership of a second vertex at a double vantage point. Additionally, the leader must react to four proxy related events: truncation of a `retracting cell`, addition/removal of PTVUIDs to a cell's Wait_Set, cell status advancement upon proxy tour completion, or labeling phantom walls upon cell deletion. Note that all vantage points are selected according to the same *parity-based vantage point selection scheme* used in the incremental partition algorithm of Sec. 2.5.

In PROPAGATE_SPARSE_VANTAGE_POINT_INFORMATION() (Table 2.9), the leader agent propagates vantage point labels and swaps fertile cells with other leaders in such a way that leader agents eventually are positioned only at sparse vantage points.

### 2.6.2 Proxy Behavior

The `proxy` portion of the internal state transition thread on lines 17-21 of Table 2.6 runs one of two subroutines depending on the status of the proxied cell: PROXY_RETRACTING_CELL() and PROXY_STABLE_CELL(). Suppose an agent $i$ is proxying for a cell $c_\xi$ in leader agent $j$'s memory. Then agent $i$ keeps a local copy of $c_\xi$ in $c^{[i]}_{\xi_{\text{proxied}}}$ and modifies it during the proxy tour. Agent $j$ updates $c_\xi$ to match $c^{[i]}_{\xi_{\text{proxied}}}$ whenever a change occurs. In PROXY_RETRACTING_CELL() (Table 2.10), the agent traverses the gap edges of $c^{[i]}_{\xi_{\text{proxied}}}$ while truncating its boundary at any encountered fertile cells in branch conflict. The goal is for the retracting proxied cell to not be in branch conflict with any fertile cells by the end of the proxy tour when its status is advanced to `stable`. If the proxy agent encounters a stable cell, say $c_{\xi'}$, in branch conflict, it must pause its proxy tour,

i.e., pause motion, until $c_{\xi'}$ becomes fertile or deleted. If the proxy were not to pause, then it would run the risk of the stable cell becoming fertile after the opportunity for the proxy to perform truncation had already passed. The pausing is accomplished by adding $\xi'$ to the cell field $c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set read by the navigation thread. Once the proxy tour is over, the leader of the proxied cell advances the cell's status to `stable` and the proxy agent enters a new mode.

In PROXY_STABLE_CELL() (Table 2.11), the goal is for the stable proxied cell to not be in branch conflict with any other stable cells by the end of the proxy tour if its status is to be advanced to `fertile`. To this end, the agent traverses the gap edges of $c^{[i]}_{\xi_{\text{proxied}}}$ while comparing $\xi^{[i]}_{\text{proxied}}$ with the PTVUID of every encountered stable cell in branch conflict with $c^{[i]}_{\xi_{\text{proxied}}}$. If a stable cell with PTVUID less than $\xi^{[i]}_{\text{proxied}}$ is encountered, then a phantom wall is labeled at the parent gap edge of $c^{[i]}_{\xi_{\text{proxied}}}$ (from this point on all other cells treat the cell as already deleted because the phantom wall at the parent gap edge serves as a deletion indicator) and the proxy agent heads straight back to the parent gap edge where it will end the proxy tour and enter `explore` mode. If the proxy agent encounters a retracting cell, say $c_{\xi'}$, in branch conflict, it must pause its proxy tour, i.e., pause motion, until $c_{\xi'}$ becomes stable or truncated out of branch conflict. If the proxy were not to pause, then it would run the risk of the retracting cell becoming stable after the opportunity for the proxy to perform deconfliction had already passed. The pausing is accomplished by adding $\xi'$ to the cell field $c^{[i]}_{\xi_{\text{proxied}}}$.Wait_Set read by the navigation thread. Note that the use of PTVUID total ordering (Definition 2.2) on line 8 of PROXY_RETRACTING_CELL() and line 9 of PROXY_STABLE_CELL() precludes the possibility of a deadlock situation where a stable and retracting cell are indefinitely waiting for each other.

Finally, if a stable cell with PTVUID less than $\xi^{[i]}_{\text{proxied}}$ is never encountered, then the leader of the proxied cell advances the cell's status to `fertile` and the proxy agent enters `explore` mode.

### 2.6.3 Explorer Behavior

The `explore` portion of the internal state transition thread on lines 22-23 of Table 2.6 consists of a single subroutine EXPLORE() shown in Table 2.12. Of all agent modes, `explore` behavior is the simplest because all the agent has to do is navigate $\mathcal{T}_{\mathcal{P}}$ in depth-first order (see Fig. 2.9) until a leader agent assigns them to become a leader at an unexplored gap edge or to perform a proxy task. The local shortest paths (lines 6 and 10) can be computed quickly and easily by the visibility graph method [80]. If the current cell that an explorer agent is visiting is ever deleted because of branch deconfliction, the explorer simply moves up $\mathcal{T}_{\mathcal{P}}$ and continues depth-first searching. By having each agent use a different gap edge ordering for the depth-first search, the deployment tends to explore many partition tree branches in parallel and thus converge more quickly. In our simulations (Sec. 2.6.5), we had each agent cyclically shift their gap edge ordering by their UID, subject to the following restriction important for proving an upper bound on number of required agents in Theorem 2.3.

**Remark 2.1** (Restriction on Depth-First Orderings)**.** *Each agent in an execution of the distributed deployment may search $\mathcal{T}_{\mathcal{P}}$ depth-first using any child ordering as long as every pair of child vertices adjacent at a double vantage point are visited in the same order by every agent.*

### 2.6.4 Performance Analysis

The convergence properties of the Distributed Depth-First Connected Deployment Algorithm shown of Table 2.6 are captured in the following theorems.

**Theorem 2.3** (Convergence)**.** *Suppose that $N$ agents are initially colocated at a common point $p_\emptyset \in V_\mathcal{E}$ of a polygonal environment $\mathcal{E}$ with $n$ vertices and $h$ holes. If the agents operate according to the Depth-First Connected Deployment Algorithm of Table 2.6, then*

*(i) the agents' visibility graph $\mathcal{G}_{\text{vis}}\mathcal{E}(P)$ consists of a single connected component at all times,*

*(ii) there exists a finite time $t^*$, such that for all times greater than $t^*$ the set of vertices in the distributed representation of the partition tree $\mathcal{T}_\mathcal{P}$ remains fixed,*

*(iii) if the number of agents $N \geq \lfloor \frac{n+2h-1}{2} \rfloor$, then for all times greater than $t^*$ every point in the environment $\mathcal{E}$ will be visibile to some agent, and there will be no more than $h$ phatom walls, and*

*(iv) if $N > \lfloor \frac{n+2h-1}{2} \rfloor$, then for all times greater than $t^*$ every cell in the distributed representation of $\mathcal{T}_\mathcal{P}$ will have* `fertile` *status and there will be precisely $h$ phantom walls.*

*Proof.* We prove the statements in order. Nonleader agents, as we have defined their behavior, remain at all times within line of sight of at least one leader agent. Leader agents likewise remain in the kernel of their cell(s) of responsibility and within line of sight of the leader agent responsible for the corresponding parent cell(s). Given any two agents, say $i$ and $j$, a path can thus be constructed by first following parent-child visibility links from agent $i$ up to the leader agent responsible for the root, then from the leader agent responsible for the root down to agent $j$. The agents' visibility graph must therefore consist of a single connected component, which is statement (i).

For statement (ii), we argue similarly to the proof of Theorem 2.1(i). During the deployment, cells are constructed only at unexplored gap edges. A cell either (1) advances though a finite number of status changes or (2) it is deleted during a proxy tour. Either way, each cell is only modified a finite number of times and only one cell is ever created at any particular unexplored gap edge. Since unexplored gap edges are diagonals of the environment and there are only finitely many possible diagonals, we conclude the set of vertices in the distributed representation of $\mathcal{T}_{\mathcal{P}}$ must remain fixed after some finite time $t^*$.

For statement (iii), we rely on an invariant: during the distributed deployment algorithm, at least two unique triangles can be assigned to every leader agent which has at least one cell of responsibility, other than the root cell, in its memory; at least one unique triangle can be assigned to the leader agent which has the root cell in its memory. One of the triangles is in a leader's own cell (primary or secondary) and its existence is ensured by the leader behavior in Table 2.7. The second triangle is in a parent cell of a cell in the agent's memory. The existence of this second triangle is ensured by the depth-first order restriction stipulated in Remark 2.1 together with the parity-based vantage point selection scheme. Remembering that the maximum number of triangles in any triangulation is $n+2h-2$ and arguing precisely as we did for the sparse vantage point locations in the proof of Theorem 2.2(iii), we find the number of agents required for full coverage can be no greater than $\lfloor \frac{n+2h-1}{2} \rfloor$. As in the proof of Theorem 2.1(v), the number of phantom walls can be no greater than $h$ because if it where then some cell would be topologically isolated.

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

**Remark 2.2** (Near Optimality without Holes)**.** *As mentioned in Sec. 2.1 at the beginning of this chapter,* $(n-2)/2$ *guards are always sufficient and occasionally necessary for visibility coverage of any polygonal enironment without holes.*

*This means that when $h = 0$, the bound on the number of sufficient agents in Theorem 2.3 statement (iii) differs from the worst-case optimal bound by at most one.*

**Theorem 2.4** (Time to Convergence)**.** *Let $\mathcal{E}$ be an environment as in Theorem 2.3. Assume time for communication and processing are negligible compared with agent travel time and that $\mathcal{E}$ has uniformly bounded diameter as $n \to \infty$. Then the time to convergence $t^*$ in Theorem 2.3 statement (ii) is $\mathcal{O}(n^2 + nh)$. Moreover, if the maximum perimeter length of any vertex-limited visibility polygon in $\mathcal{E}$ is uniformly bounded as $n \to \infty$, then $t^*$ is $\mathcal{O}(n + h)$.*

*Proof.* As in the proof of Theorem 2.3, every cell which is never deleted has at least one unique triangle and there are at most $n+2h-2$ triangles total, therefore there are at most $n+2h-2$ cells which are never deleted. The maximum number of phantom walls ever created is $h$ (Theorem 2.3). Since cells are only ever deleted when a phantom wall is created, at most $h$ cells are ever deleted. Summing the bounds on the number cells which are and are not deleted, we see the total number of cells any agent must ever visit during the distributed deployment is $n + 2h - 2 + h = n + 3h - 2$. Let $l_{\mathrm{d}}$ be the maximum diameter of any vertex-limited visibility polygon in $\mathcal{E}$. Then, neglecting time for proxy tours, an agent executing depth-first search on $\mathcal{T}_{\mathcal{P}}$ will visit every vertex of $\mathcal{T}_{\mathcal{P}}$ in time at most $2u_{\max}l_{\mathrm{d}}(n+3h-2)$. Now Let $l_{\mathrm{p}}$ be the maximum perimeter length of any vertex-limited visibility polygon in $\mathcal{E}$. Then the total amount of time agents spend on proxy tours, counting two tours for each cell, is $2u_{\max}l_{\mathrm{p}}(n + 3h - 2)$. Exploring and leading agents operate in parallel and at most every agent waits for every proxy tour, so it must be that

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

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

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

**Lemma 2.3** (Memory and Communication Complexity)**.** *Let $k$ be the maximum number of vertices of any vertex-limited visibility polygon in the environment $\mathcal{E}$ and suppose $\mathcal{E}$ is represented with fixed resolution. Then the required memory size for an agent to run the distributed deployment algorithm is $\mathcal{O}(Nk)$ bits and the message size is $\mathcal{O}(k)$ bits.*

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

### 2.6.5 Simulation Results

We used C++ and the VisiLibity library [81] to simulate the Distributed Depth-First Deployment Algorithm of Table 2.6. An example simulation run is shown in Fig. 2.1 for an environment with $n = 41$ vertices and $h = 4$ holes. The environment was fully cover in finite time by only 13 agents, which indeed is less than the upper bound $\lfloor \frac{n+2h-1}{2} = 24 \rfloor$ given by Theorem 2.3.

### 2.6.6 Extensions

There are several ways that the distributed deployment algorithm can be directly extended for robustness to agent arrival, agent failure, packet loss, and removal of an environment edge. Robustness to agent arrival can be achieved by having any new agents simply enter `explore` mode, setting $\xi^{[i]}_{\text{current}}$ to be the PTVUID of the first cell they land in, and setting $\xi^{[i]}_{\text{last}}$ to be the parent PTVUID

of $\xi_{\text{current}}$. The line-of-sight connectivity guaranteed by Theorem 2.3 allows single-agent failures to be detected and handled by having the visibility neighbors of a failed agent move back up the partition tree as necessary to patch the hole left by the failed agent. For robustness to packet loss, agents could add a receipt confirmation and/or parity check protocol. If a portion of the environment were blocked off during the beginning of the deployment but then were revealed by an edge removal (interpreted as the "opening of a door"), the deployment could proceed normally as long as the deleted edge were marked as an `unexplored` gap edge in the cell it belonged to.

Less trivial extensions include (1) the use of distributed assignment algorithms such as [82, 83] for guiding explorer agents to tasks faster than depth-first search, or (2) performing the deployment from multiple roots, i.e., when different groups of agents begin deployment from different locations. Deployment from multiple roots can be achieved by having the agents tack on a root identifier to their PTVUID, however, it appears this would increase the bound on number of agents required in Theorem 2.3 by up to one agent per root.

## 2.7 Conclusion

In this chapter we have presented the first distributed deployment algorithm which solves, with provable performance, the Distributed Visibility-Based Deployment Problem with Connectivity in polygonal environments with holes. We began by designing a centralized incremental partition algorithm, then obtained the distributed deployment algorithm by asynchronous distributed emulation of the centralized algorithm. Given at least $\lfloor \frac{n+2h-1}{2} \rfloor$ agents in an environment with

$n$ vertices and $h$ holes, the deployment is guaranteed to achieve full visibility coverage of the environment in time $\mathcal{O}(n^2 + nh)$, or time $\mathcal{O}(n + h)$ if the maximum perimeter length of any vertex-limited visibility polygon in $\mathcal{E}$ is uniformly bounded as $n \to \infty$. The deployment behaved in simulations as predicted by the theory and can be extended to achieve robustness to agent arrival, agent failure, packet loss, removal of an environment edge (such as an opening door), or deployment from multiple roots.

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

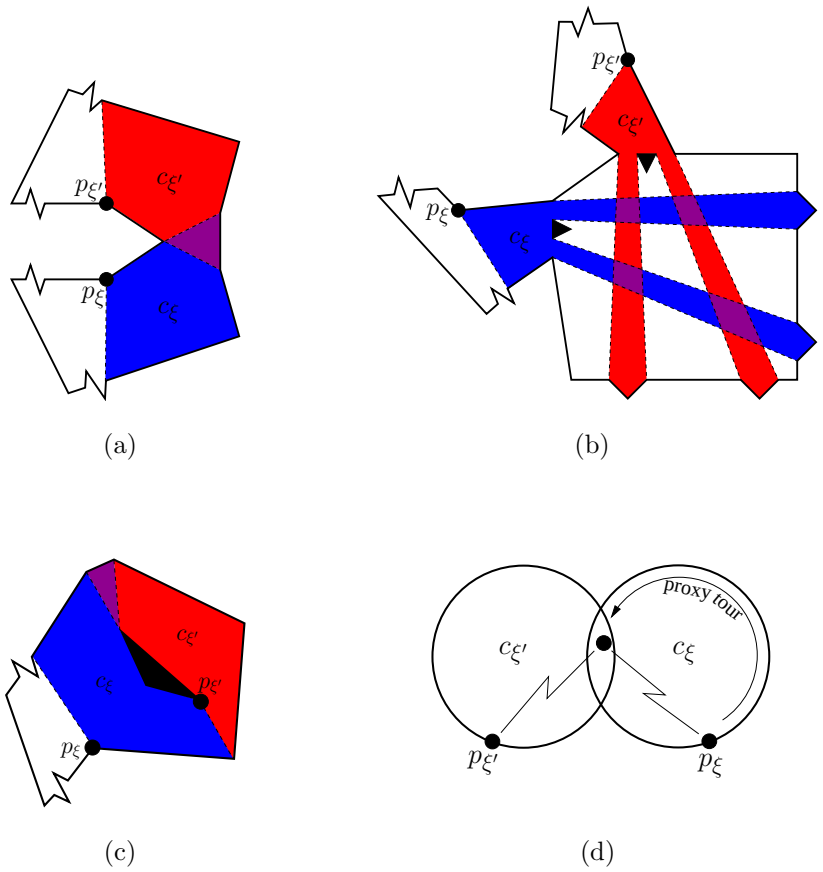



Figure 2.5. (a,b,c) The incremental partition algorithm of Table 2.1 and distributed deployment algorithm of Table 2.6 may discard a cell $c_\xi$ if it is in *branch conflict* with another cell $c_{\xi'}$ already in the partition tree, i.e., when $c_\xi$ and $c_{\xi'}$ and are not openly disjoint. In these three examples, blue represents one cell $c_\xi$, red another cell $c_{\xi'}$, and purple their intersection $c_\xi \cap c_{\xi'}$. A cell can even conflict with it's own parent if they enclose a hole as in (c). (d) In the distributed deployment algorithm, leader agents position themselves at partition tree vantage points ($p_\xi$ and $p_{\xi'}$) and retain in their memory the corresponding cells ($c_\xi$ and $c_{\xi'}$). Even if the leader agents are not mutally visible, their cells may intersect as shown abstractly by the Venn diagram. Sending a proxy agent, on a *proxy tour* around one of the cell boundaries guarantees it will enter the cells' intersection so that communication between leaders can be proxied. The leaders can then establish a local common reference frame and compare cell boundaries in order to solve branch conflicts.

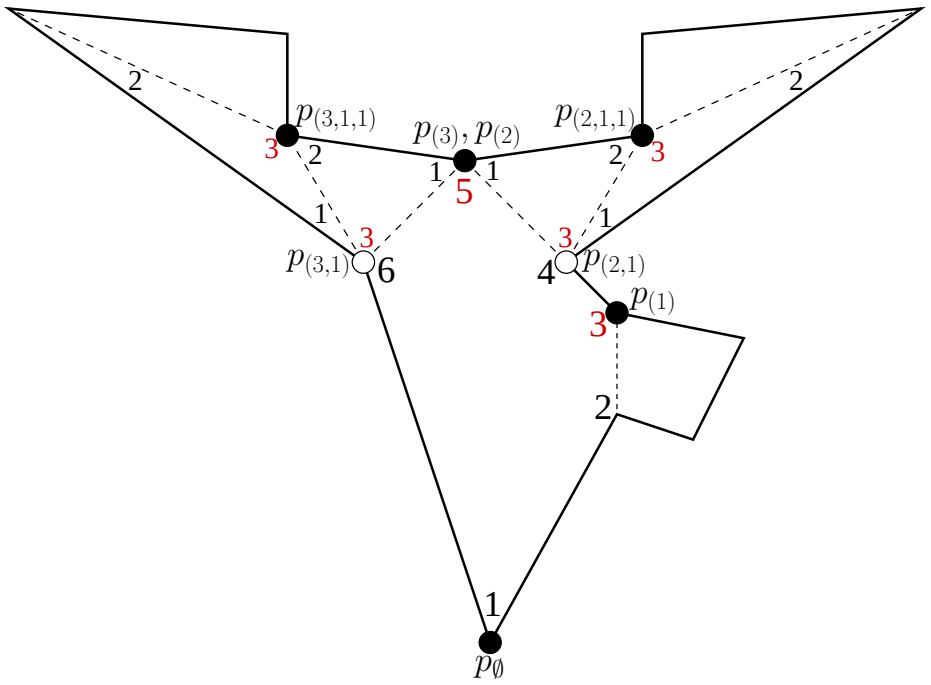
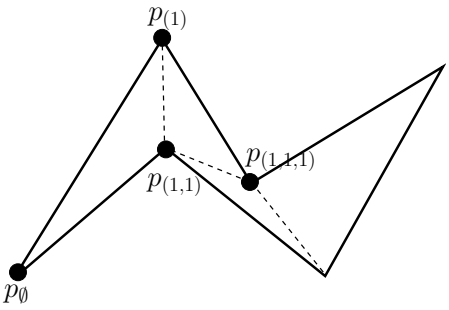
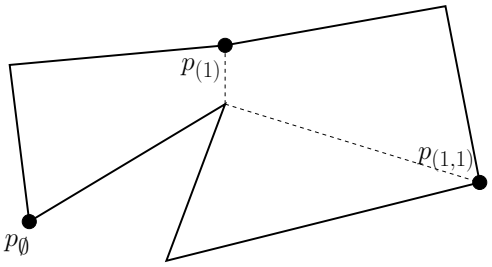



Figure 2.6. The example used in Fig. 2.3 showed a typical incremental partition in which there were neither double vantage points nor any triangular cells. This example, on the other hand, shows these special cases. Disks, black or white, show vantage points produced by the incremental partition algorithm of Table 2.1. Integers show enumerations of the cells used for the *parity-based vantage point selection scheme*. The double vantage points $p_{(2)}$ and $p_{(3)}$ are colocated. The cells $c_{(2)}$, $c_{(3)}$, $c_{(2,1)}$, $c_{(3.1)}$, $c_{(2,1,1)}$, and $c_{(3,1,1)}$ are triangular. The vantage points colored black are the *sparse vantage points* found by the postprocessing algorithm of Table 2.3. Under the distributed deployment algorithm of Table 2.6, robotic agents position themselves at sparse vantage points.




(a)




(b)

Figure 2.7. (a) An example of when the final number of vantage points in $\mathcal{T}_{\mathcal{P}}$ is equal to the upper bound $n + 2h - 2$ given in Theorem 2.1. (b) An example of when the number of points in $\mathbb{R}^2$ where at least one sparse vantage point is located is equal to the upper bound $\left\lfloor \frac{n+2h-1}{2} \right\rfloor$ given in Theorems 2.2 and 2.3.


(a) (b)

Figure 2.8. In the distributed deployment algorithm of Table 2.6, each agent may switch between `lead`, `proxy`, and `explore` mode based on certain asynchronous events. Leaders are responsible for maintaining a distributed representation of the partition tree $\mathcal{T}_{\mathcal{P}}$, proxy agents help establish communication for solving branch conflicts (as in Fig. 2.5d), and explorers systematically navigate through $\mathcal{T}_{\mathcal{P}}$ in search of opportunities to become a leader or proxy (as in Fig. 2.9). (b) Any cell in a leader's memory has a status which takes the value `retracting`, `stable`, or `fertile`. Each cell status is initially `retracting`. The status of a `retracting` cell is advanced to `stable` after the execution of a proxy tour in which the cell is truncated as necessary to ensure no branch conflict with any `fertile` cells. In a second proxy tour, a `stable` cell is either deleted or advanced to `fertile` status depending on whether it is found to be in branch conflict with another `stable` cell of smaller PTVUID (according to total ordering Def. 2.2). Only when a cell has attained `fertile` status can any child cells be added at its unexplored gap edges.


Figure 2.9. In the distributed deployment algorithm of Table 2.6, explorer agents search the partition tree $\mathcal{T}_{\mathcal{P}}$ depth-first for leader or proxy tasks they could perform. An agent in a cell, say $c_\xi$, can always order the gap edges of $c_\xi$, e.g., counterclockwise from the parent gap edge $g_\xi$. The depth-first search progresses by the agent always moving to the next unvisited child or unexplored gap edge in that ordering. The agent thus moves from cell to cell deeper and deeper until a leaf (a vertex with no children) is found. Once at a leaf, the agent backtracks to the most recent vertex with unvisited child or unexplored gap edges and the process continues. As an example, (left) shows the depth-first order an agent would visit the vertices of $\mathcal{T}_{\mathcal{P}}$ in Fig. 2.3f if the gap edges in each cell were ordered couterclockwise from the parent gap edge. If the agent instead uses a gap edge ordering cyclically shifted by one, then (right) shows the different resulting depth-first order. If each agent uses a different gap edge ordering, e.g., cyclically shifted by their UID, then different branches of $\mathcal{T}_{\mathcal{P}}$ are explored in parallel and the deployment tends to cover the environment more quickly.

# Chapter 3

# Centralized Searchlight Scheduling[1]

## 3.1 Introduction

Consider a group of point guards statically positioned in a nonconvex polygonal environment with holes, e.g., a floor plan. Each guard is equipped with a single *searchlight*, a ray sensor which can rotate about the guard's position but cannot penetrate the boundary of the environment (imagine a ray of light such as a laser range finder, or a camera with a very narrow field of view). A searchlight aims only in one direction at a time and cannot penetrate the boundary of the environment, but its direction can change continuously. A point is detected by a searchlight at some instant if and only if the point lies on the ray. Targets are points which move arbitrarily fast. The *Searchlight Scheduling Problem* is to

> Find a schedule to rotate a set of stationary searchlights such that any target in an environment will necessarily be detected in finite time.

[1] **Reprinted from [84] with permission of World Scientific Publishing Company.**

A searchlight problem instance consists of an environment together with a set of stationary sensor positions. Obviously there can only exist a search schedule if all points in the environment are visible from some guard. For a graphical description of our objective see Fig. 3.1.

To our knowledge the Searchlight Scheduling Problem was first introduced by Sugihara, Suzuki and Yamashita.[25] They give a solution, the "One Way Sweep Strategy", to the limited class of searchlight scheduling problem instances in which the environment is simply connected and there is at least one searchlight located on the boundary for every connected component of their visibility graph. In [26] an upper bound is given on the number of guards with multiple searchlights sufficient in polygonal environments containing holes. We adopt the convention in [26] and call a mobile guard possessing $k$ searchlights a *$k$-searcher*. Some articles involving 1-searchers, sometimes calling them *flashlights* or *beam detectors*, are [27], [28], [29], and [30]. Closely related is the Classical Art Gallery Problem, namely that of finding a minimum set of guards (with omnidirectional vision) such that the entire polygon is visible. There are many variations on the art gallery problem which are wonderfully surveyed in [6], [7], and [8]. With an emphasis on practical imaging considerations, [16] describes a centralized task-specific procedure for choosing the locations of cameras in a network. As described in [22] and [23], there are distributed algorithms for deploying guards into simple polygons such that their final positions are a solution to the art gallery problem. We later used these ideas in [85] for distributed deployment of agents such that the agents are able to execute a searchlight schedule in an asynchronous distributed manner from their final positions.

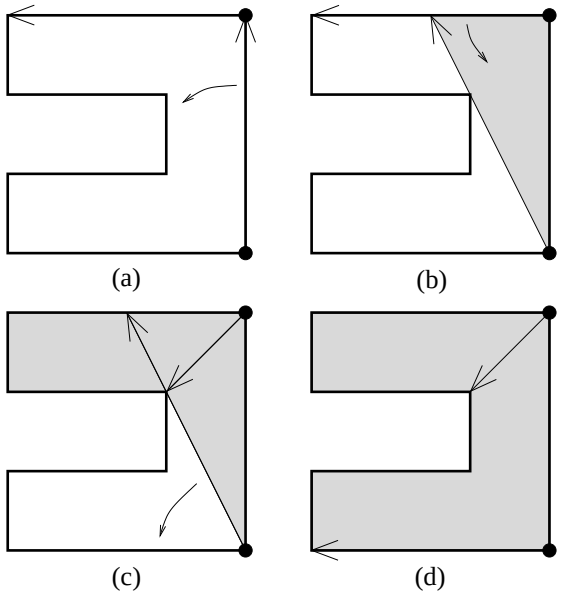



Figure 3.1. A simple example of a searchlight schedule. "Clear" regions where no undetected evader could exist are shown in gray. From (a) to (d): First the lower searchlight aims at the upper searchlight and sweeps until it hits a corner where its visibility is occluded. Next, the upper searchlight sweeps the area the lower searchlight cannot see. Finally, the lower searchlight continues sweeping the remainder of the environment. No target, no matter how fast, would be able to avoid detection by this rotation sequence.

Exact cell decomposition, a method we use in the present chapter, has been used in the design of complete algorithms to solve visibility-based pursuit-evasion problems before, e.g., in [32] and [27]. In [32] an algorithm is given for a single mobile searcher with omnidirectional vision, and it is shown that determining the minimum number of such pursuers required to clear a polygonal environment with holes is NP-hard. In [27] a complete algorithm is described for a single mobile "$\phi$-searcher" having an angle $\phi$ field of view, and it is shown that determining the minimum number of such pursuers required to clear a polygonal environment with holes is also NP-hard. Due to anticipated computational complexity, both

articles, perhaps appropriately, dismiss the idea of using a complete exact cell decomposition for the case of multiple searchers, although they do implement incomplete extensions which they claim work well for practical purposes. It is suggested on page 569 of [62] that implementation of a complete exact cell decomposition algorithm for multiple pursuers would be further complicated because "some of the cell boundaries are algebraic surfaces due to complicated interactions between the visibility polygons of different pursuers." To our knowledge nobody has carried out the design of a complete algorithm to solve any visibility-based pursuit-evasion problem involving arbitrary polygonal environments with holes. However, there are at least two noteworthy articles involving multiple pursuers in polygonal environments without holes. In [41] a polynomial time complete algorithm is provided for two 1-searchers in a simple polygonal environment, but has not been extended to three or greater pursuers and it is not clear how to do so. In [42] a polynomial time complete algorithm is given to determine the minimum number of $\infty$-searchers (omnidirectional vision) necessary to clear a simple polygon, but under the constraints that (1) the pursuers are in a chain configuration where consecutive pursuers along the chain are mutually visible, and (2) end pursuers must remain on the polygon boundary.

There are three main contributions in this chapter. First, we show by exact cell decomposition that if an instance of the Searchlight Scheduling Problem permits any solution at all, then it also permits a solution in a reduced discrete solution space. The second contribution is to use the knowledge of the solution space discretization to design a complete[2] algorithm for searchlight scheduling. Although it remains an open problem whether searchlight scheduling is NP-hard,

[2]Here *complete* means that if a solution exists, the algorithm is guaranteed to find one in finite time.

our computed examples demonstrate that for searchlights, even in environments with holes, the time complexity of a complete exact cell decomposition is not entirely prohibitive and can be practical for problem instances of useful size. To accomplish this, we construct our cell decomposition dependent on the pursuer positions so that there is no need to explicitly compute any algebraic surfaces. At this time no other algorithm exists to solve the general Searchlight Scheduling Problem. As a third contribution we treat a new problem which we call the *$\phi$-Searchlight Scheduling Problem* in which *$\phi$-searchlights* sense not just along a ray, but over a finite field of view (see Fig 3.12). We show how our searchlight scheduling algorithm can be extended to take advantage of $\phi$-searchlights having a wider field of view than just a ray. This is an important extension because for cameras having a finite field of view it is a much more realistic sensor model. We envision our algorithms and/or other algorithms inspired by this work will one day be used in automating the design of security systems consisting of networks of statically positioned rotating sensors and actuators.

This chapter is organized as follows. Section 3.2 covers preliminary notation, technical definitions, and statement of assumptions. Section 3.3 provides an extensive theoretical development which culminates in a proof showing a reduction of the searchlight scheduling solution space. The solution space reduction guarantees the completeness of our algorithm, which we present in Section 3.4. In Section 3.5 we introduce the $\phi$-Searchlight Scheduling Problem and explain how our results for searchlight scheduling can be directly extended for $\phi$-searchlights. We conclude in Section 3.6.

## 3.2 Preliminaries

### 3.2.1 Notation

We begin by introducing some basic notation. We let $\mathbb{R}$ and $\mathbb{T}^d$ represent the set of real numbers and the $d$-dimensional torus, respectively. Clockwise is abbreviated by cw, and counterclockwise by ccw. Given two points $a, b \in \mathbb{R}^2$, we let $[a, b]$ signify the *closed segment* between $a$ and $b$. Similarly, $]a, b[$ is the *open segment* between $a$ and $b$, $[a, b[$ represents the set $]a, b[\cup\{a\}$ and $]a, b]$ is the set $]a, b[\cup\{b\}$. Given a set (resp. list) $A$, $|A|$ denotes the cardinality of the set (resp. list), $A^\circ$ the interior, $\bar{A}$ the closure, and $\partial A$ the boundary. Also, we shall use $P$ to refer to tuples of elements in $\mathbb{R}^2$ of the form $(p^{[1]}, \ldots, p^{[N]})$ (these will be the locations of the searchlights), where $N$ denotes the total number of searchlights.

We turn our attention to the environments we are interested in and to the concepts of visibility in such environments. The environment, denoted by $\mathcal{E}$, is closed and consists of an (outer) polygon containing polygonal holes. The boundaries of these polygons do not intersect themselves or each other. Throughout this chapter, $n$ will refer to the number of edges of $\mathcal{E}$ (holes included) and $r$ the number of reflex vertices. A reflex vertex is constituted by any concave vertex of the outer polygon or convex vertex of a hole. A point $q \in \mathcal{E}$ is *visible from* $p \in \mathcal{E}$ if $[p, q] \subset \mathcal{E}$. The *visibility set* $\mathcal{V}(p) \subset \mathcal{E}$ from a point $p \in \mathcal{E}$ is the set of points in $\mathcal{E}$ visible from $p$. A *visibility gap* of a point $p \in \mathcal{E}$ is defined as any line segment $[a, b]$ such that $]a, b[ \subset \operatorname{int} \mathcal{E}$, $[a, b] \subset \partial\mathcal{V}(p)$, and it is maximal in the sense that $a, b \in \partial\mathcal{E}$. Intuitively, visibility gaps form the border between portions of $\mathcal{E}$ visible from $p$ and portions of $\mathcal{E}$ not visible from $p$.

Now we introduce some notation and definitions specific to the Searchlight

Scheduling Problem. An instance of the Searchlight Scheduling Problem is specified by a pair $(\mathcal{E}, P)$, where $\mathcal{E}$ is an environment and $P$ is a set of searchlight locations in $\mathcal{E}$. For convenience, we will refer to the $i$th searchlight as $s^{[i]}$ (which is located at $p^{[i]} \in \mathbb{R}^2$), and $S = \{s^{[0]}, \ldots, s^{[N-1]}\}$ will be the set of all searchlights. We let $\theta^{[i]}$ denote the configuration angle of the searchlight in radians from the positive horizontal axis, and $\Theta = \{\theta^{[0]}, \ldots, \theta^{[N-1]}\}$ denote the joint configuration. So, if we say, e.g., aim $s^{[i]}$ at point $e$, what we really mean is set $\theta^{[i]}$ equal to an angle such that the $i$th searchlight is aimed at $e$. Note that searchlights do not block visibility of other searchlights.

The next few definitions are similar to those in [25].

**Definition 3.1** (Schedule)**.** *Let $[0, T]$ be a finite interval of real time. A* schedule *of a searchlight $\theta^{[i]}(t) \in \Theta(t)$ is a continuous function $\theta^{[i]} : [0, T] \to \mathbb{T}^1$ such that $s^{[i]}$ changes direction of rotation at most a finite number of times.*

The requirement that no searchlight switches direction of rotation an infinite number of times is important for practical realizability.

**Definition 3.2** (Active)**.** *$s^{[i]}$ is* active *at time $t$ if it is rotating (has nonzero angular velocity), otherwise it is* inactive*.*

The *ray* of $s^{[i]}$ at time $t \in [0, T]$ is the intersection of $\mathcal{V}(p^{[i]})$ and the semi-infinite ray starting at $p^{[i]}$ with direction $\theta^{[i]}(t)$. $s^{[i]}$ is said to be *aimed* at a point $e \in \mathcal{E}$ at some time instant if $e$ is on the ray of $s^{[i]}$. A point $e$ is *illuminated* if there exists a searchlight aimed at $e$.

**Definition 3.3** (Separability)**.** *Two points in $\mathcal{E}$ are* separable *at time $t \in [0, T]$ if every continuous path connecting them in the interior of $\mathcal{E}$ contains an illuminated point, otherwise they are* nonseparable*. Two regions $R_1$ and $R_2$ in $\mathcal{E}$ are* separable *if every pair of points $e_1 \in R_1$ and $e_2 \in R_2$ are separable.*

**Definition 3.4** (Contamination and Clarity)**.** *A point $e \in \mathcal{E}$ is* contaminated *at time zero if and only if it is not illuminated. The point $e$ is contaminated at time $t \in ]0, T]$ if and only if there exists a continuous function $f : [0, t] \to \mathcal{E}$ such that $f(t) = e$ and for every instant $t' \in [0, t]$, $f(t')$ is not illuminated by any searchlight. A point which is not contaminated is called* clear*. A region is said to be* contaminated *if it contains a contaminated point, otherwise it is* clear*.*

**Definition 3.5** (Search Schedule)**.** *Given $\mathcal{E}$ and a set of searchlight locations $P = \{p^{[0]}, \ldots, p^{[N-1]}\}$, a schedule $\Theta(t) = \{\theta^{[0]}(t), \ldots, \theta^{[N-1]}(t)\} : [0, T] \to \mathbb{T}^N$ is a* search schedule for $(\mathcal{E}, P)$ *if $\mathcal{E}$ is clear at $T$.*

### 3.2.2 Assumptions

The following *main assumptions* will be made about every problem instance in this chapter:

(i) The environment is static and has a finite number of vertices.

(ii) Every point in the environment is visible from some searchlight and there is a finite number $N$ of searchlights.

*Comments:* If there were some point in the environment not visible from any searchlight, then a target could remain there undetected for all time.

(iii) No two searchlights are co-located.

(iv) All searchlights are switched on at all times, even when rotating.

*Comments:* Leaving inactive searchlights switched on can only increase, and not decrease the chance of detecting an evader. One might argue that leaving a sensor switched on could be costly, e.g., from an energy standpoint, but we are not considering such things in our solution.

## 3.3 Reducing the Solution Space

The solution space of the Searchlight Scheduling Problem is the set of all possible schedules. This section focuses on defining several special classes of schedules and showing that the existence of a search schedule in the most general continuous class implies the existence of a search schedule in a reduced discrete class which can be searched for a solution. This is accomplished by an exact cell decomposition of the searchlights' joint configuration space ($\mathbb{T}^N$). Our algorithm in Section 3.4 and its completeness will follow directly from the solution space reduction.

At any instant of a schedule, searchlights are aimed in various directions so that their beams separate (in the sense of Definition 4.2) the environment $\mathcal{E}$ into a set of distinct polygonal regions (possibly containing holes) each of which is either entirely clear or entirely contaminated. We formalize this.

**Definition 3.6** (Maximal Nonseparable Region)**.** *For a fixed searchlight configuration $\Theta$ and an unilluminated point $e \in \mathcal{E}$, the equivalence class of all points in $\mathcal{E}$ which are nonseparable from $e$ is called a* maximal nonseparable region*.*

At any time during a schedule there is a finite number of maximal nonseparable regions. As an example, in Fig. 3.1(c) there are 4 maximal nonseparable regions, 3 of which are clear.

**Definition 3.7** (Support)**.** *If a portion of a searchlight's beam forms part of the boundary of a maximal nonseparable region, then that searchlight is said to* support *that maximal nonseparable region. The set of all searchlights whose beams form the boundary of a maximal nonseparable region is called the* support *of that maximal nonseparable region.*

The support of a maximal nonseparable region changes, in general, over the

course of a schedule. As a schedule is executed, maximal nonseparable regions, in addition to continuously deforming, may undergo any of the following changes:

*Disappear*: A maximal nonseparable region disappears if and only if its area goes to zero. This can happen if one or more searchlights rotate (i) into $\partial\mathcal{E}$, (ii) into coincidence with another searchlight's beam, (iii) onto the intersection of other searchlights' beams, or (iv) onto the intersection point of another searchlight's beam with $\partial\mathcal{E}$. The only way to clear a contaminated maximal nonseparable region is to make it disappear. Examples are shown in Fig 3.3.

*Appear*: The reverse of disappear. Note any maximal nonseparable region which appears remains clear until merging with a contaminated region.

*Merge*: Two or more maximal nonseparable regions can merge into one if a searchlight rotates (i) past a reflex vertex of $\partial\mathcal{E}$ where its visibility is occluded, (ii) away from a reflex vertex which it was grazing, or (iii) past another searchlight not on $\partial\mathcal{E}$. Examples are shown in Fig. 3.3. A clear maximal nonseparable region can become contaminated only by merging with a contaminated region.

*Split*: The reverse of merge.

Indeed, any possible change to a maximal nonseparable region will fall under one of the above categories. To describe the entire system evolution precisely, we say at time $t$ it possesses an *information state* which consists of the searchlights' joint configuration $\Theta(t) \in \mathbb{T}^N$ together with the *contamination state* $C(t)$ of the environment $\mathcal{E}$. By contamination state is meant an encoding of which points in $\mathcal{E}$ are contaminated, e.g., a binary labeling of the maximal nonseparable regions

(0 for clear, 1 for contaminated). We denote the information state by a pair $(\Theta(t), C(t))$, or by $(\Theta, C)$ when the time is implicitly understood. Note, however, that the concept of information state has no intrinsic dependence on time, so we may also speak of the information state of a searchlight system without it being associated with any particular schedule. The information state takes on a value in a continuous information space $\mathcal{I}$. To every searchlight schedule corresponds a unique trajectory through $\mathcal{I}$, thus a search schedule can simply be viewed as an information space trajectory which begins with a completely contaminated information state and ends with a completely clear information state. Ultimately we will discretize the continuous information space $\mathcal{I}$ into a so-called *information graph* $\mathcal{G}_{\mathcal{I}}$ which can be searched systematically for a searchlight rotation schedule. We delay discussing the information graph further until Section 3.4.

The definitions to follow will allow us to describe the exact cell decomposition of the searchlight configuration space ($\mathbb{T}^N$).

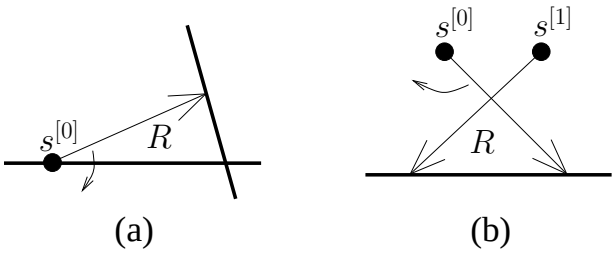



Figure 3.2. How a maximal nonseparable region $R$ can be made to disappear by a single searchlight $s^{[0]}$ rotating as indicated in each case by the smaller arrow. The thick line segments may represent either portions of $\partial\mathcal{E}$ or other searchlight beams. In this way, (a) could depict $s^{[0]}$ rotating either into $\partial\mathcal{E}$, or into coincidence with another searchlight's beam. Likewise, (b) could depict $s^{[0]}$ rotating either onto the intersection of other searchlights' beams, or onto the intersection point of another searchlight's beam with $\partial\mathcal{E}$.

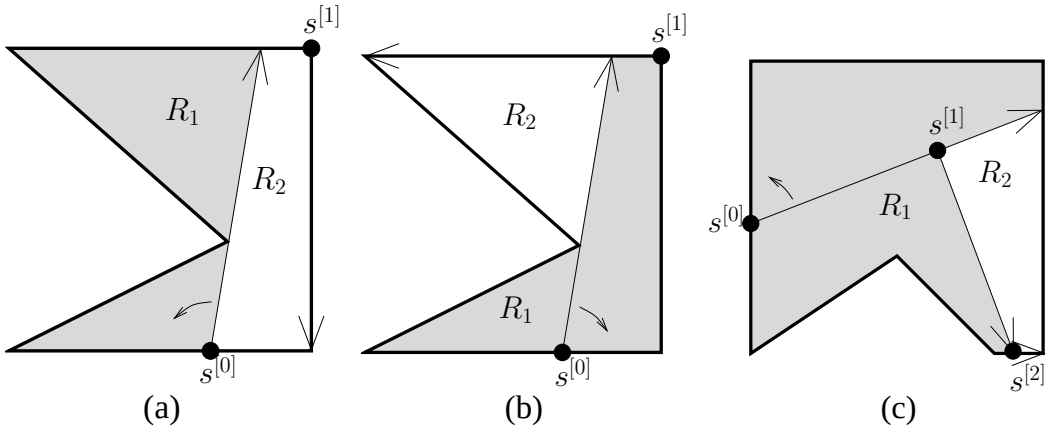



Figure 3.3. The only manner in which two or more maximal nonseparable regions can merge into one is by some searchlight rotating either (a) past a reflex vertex of $\partial\mathcal{E}$ where its visibility is occluded, (b) away from a reflex vertex which it was grazing, or (c) past another searchlight not on $\partial\mathcal{E}$. In each example the clear region $R_1$ will become contaminated when it merges with the contaminated region $R_2$. Note that in (a) and (b) the reflex vertex could have also been a flat edge (between two reflex vertices) aligned with $s^{[0]}$'s beam, but the effect is the same.

**Definition 3.8** (Critical Angle)**.** *An angle $\psi$ is a* critical angle[3] *of $s^{[i]}$ if $\theta^{[i]} = \psi$ implies either*

*(i) $s^{[i]}$ is located on an edge (including endpoints) of $\partial\mathcal{E}$ and is aimed along that edge,*

*(ii) $s^{[i]}$ is aimed at one of its visibility gaps,*

*(iii) $s^{[i]}$ is aimed at another visible searchlight, or*

*(iv) $s^{[i]}$ is aimed directly away from another visible searchlight.*

Basic examples of critical angles are shown by the dashed lines in Fig. 3.3.

[3] *Thanks to Howie Choset for pointing out the appropriateness of the name "critical angle". Taking the origin at a searchlight, its beam can be viewed as a level set of the Morse function $h(x, y) = \tan(y/x)$ so that the critical angles are where $h(x, y)$ has a critical point (constituted by a reflex vertex of $\partial\mathcal{E}$ or a searchlight in the interior of $\mathcal{E}$). See [86], [87].*

More complicated examples can be found in Fig. 3.9 and 3.10. A searchlight configuration is a *critical configuration* if every searchlight is aimed along one of its critical angles. An information state corresponding to searchlights in a critical configuration is a *critical information state*.

The next definition gives a useful notation for expressing the relationship between two information states with the same searchlight configuration but different contamination states.

**Definition 3.9** (Partial Ordering on Information States)**.** *Given two information states* $(\Theta_1, C_1)$ *and* $(\Theta_2, C_2)$*, we write* $(\Theta_1, C_1) \succeq (\Theta_2, C_2)$ *if* $\Theta_1 = \Theta_2$ *and every maximal nonseparable region which is clear in* $C_1$ *is also clear in* $C_2$*. If it is understood from context that* $\Theta_1 = \Theta_2$*, then we simply write* $C_1 \succeq C_2$*.*

**Definition 3.10** (Critical Intervals and Rectangles)**.** *Let* $\psi_j^{[i]}$ *and* $\psi_k^{[i]}$ *be either adjacent or equal critical angles of* $s^{[i]}$*. Then an angular interval* $\lambda^{[i]} = ]\psi_j^{[i]}, \psi_k^{[i]}[ \subset \mathbb{T}$ *(resp.* $[\psi_j^{[i]}, \psi_k^{[i]}]$*) consisting of all angles which are*

*(i) between* $\psi_j^{[i]}$ *and* $\psi_k^{[i]}$*, and*

*(ii) ccw from* $\psi_j^{[i]}$ *to* $\psi_k^{[i]}$

*is an* open critical interval *(resp.* closed critical interval*) if* $\lambda^{[i]}$ *does not contain any critical angles of* $s^{[i]}$ *(resp. other than* $\psi_j^{[i]}$ *and* $\psi_k^{[i]}$*). The angles* $\psi_j^{[i]}$ *and* $\psi_k^{[i]}$ *are the* bounding angles *of the critical interval. In the case* $\psi_j^{[i]} = \psi_k^{[i]}$*,* $\lambda^{[i]} = \{\psi_j^{[i]}\}$ *and* $\lambda^{[i]}$ *is called a* null critical interval*. The Cartesian product* $\Lambda = \lambda^{[0]} \times \cdots \times \lambda^{[N-1]} \subset \mathbb{T}^N$ *of critical intervals is a* critical rectangle*.*

**Definition 3.11** (Minimal Critical Rectangle)**.** *Given a searchlight configuration* $\Theta_0 = \{\theta_0^{[0]}, \ldots, \theta_0^{[N-1]}\}$*, the* minimal critical rectangle $\Lambda$ *containing* $\Theta_0$ *is the Cartesian product of critical intervals* $\Lambda = \lambda^{[0]} \times \cdots \times \lambda^{[N-1]}$*, where each* $\lambda^{[i]}$ *(*$i = 0, \ldots, N-1$*) is the unique smallest critical interval containing* $\theta_0^{[i]}$*. If* $\theta_0^{[i]}$ *is a noncritical angle, then* $\lambda^{[i]}$ *is open. If* $\theta_0^{[i]}$ *is critical, then* $\lambda^{[i]}$ *is a null (closed) critical interval.*

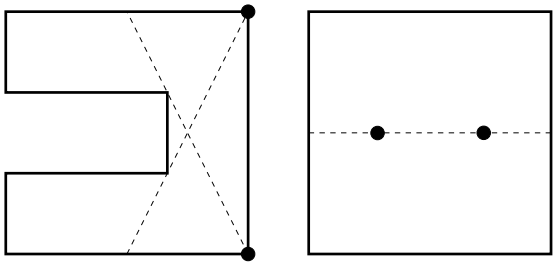



Figure 3.4. Examples of critical angles: (a) Each searchlight has two critical angles aiming along the adjacent walls and one aiming towards its visibility gap. (b) Each searchlight has two critical angles, one pointing directly toward the other searchlight and one pointing directly away.

**Remark 3.1.** *By Definition 3.11, the minimal critical rectangle* $\Lambda$ *containing a searchlight configuration* $\Theta_0$ *is not necessarily closed or open. In particular,* $\Lambda$ *is*

*(i) closed iff* $\Theta_0$ *is a critical configuration,*

*(ii) open iff in* $\Theta_0$ *no searchlight is aimed at a critical angle, or*

*(iii) neither open nor closed otherwise.*

**Lemma 3.1.** *Given a searchlight configuration* $\Theta_0$*, the minimal critical rectangle* $\Lambda$ *containing* $\Theta_0$ *is unique.*

*Proof.* Letting $\Lambda = \lambda^{[0]} \times \cdots \times \lambda^{[N-1]}$ as in Definition 3.11, uniqueness of $\Lambda$ follows immediately from the uniqueness of each $\lambda^{[i]}$ $(i = 0, \ldots, N-1)$. □

If the searchlight configuration does not leave a minimal critical rectangle, then by definition no merging can happen[4]. This gives the following lemma.

**Lemma 3.2.** *Let* $\Lambda$ *be the minimal critical rectangle containing some searchlight configuration. Then any maximal nonseparable region which is cleared without leaving* $\Lambda$ *necessarily remains clear until* $\Lambda$ *is left.*

[4]In this way, critical rectangles are reminiscent of the "conservative regions" defined in [32].

In fact we can make a slightly stronger statement as in the following two definitions and lemma.

**Definition 3.12** ($\Lambda$-equivalence of maximal nonseparable regions)**.** *Let $\Lambda$ be the minimal critical rectangle containing some searchlight configuration, and $R$ and $R'$ distinct maximal nonseparable regions present when the searchlight configuration is somewhere in $\bar{\Lambda}$. Then $R$ and $R'$ are* $\Lambda$-equivalent *if it is possible for $R$ and $R'$ to merge without the searchlight configuration leaving $\bar{\Lambda}$.*

**Definition 3.13** (modulo $\Lambda$-equivalence)**.** *Let $\Lambda$ be the minimal critical rectangle containing some searchlight configuration. When we say a property holds for maximal nonseparable regions* modulo $\Lambda$-equivalence*, we intend the following. Any $\Lambda$-equivalence class of maximal nonseparable regions, say $\{R_0, R_1, \ldots, R_{M-1}\}$, is to be interpreted as a single maximal nonseparable region, say $R$, where*

*(i) the contamination state of $R$ is taken to be the logical OR of the contamination states of $R_0, R_1, \ldots, R_{M-1}$, and*

*(ii) the area of $R$ is taken to be the sum of the areas of $R_0, R_1, \ldots, R_{M-1}$.*

Roughly put, making a statement about maximal nonseparable regions modulo $\Lambda$-equivalence amounts to ignoring the splitting and merging which can happen when the searchlight configuration moves between the minimal critical rectangle $\Lambda$ and its relative boundary $\bar{\Lambda} \setminus \Lambda$.

**Lemma 3.3.** *Let $\Lambda$ be the minimal critical rectangle containing some searchlight configuration. Then, modulo $\Lambda$-equivalence, any maximal nonseparable region which is cleared without leaving $\bar{\Lambda}$ necessarily remains clear until $\bar{\Lambda}$ is left.*

*Proof.* A maximal nonseparable region can only be recontaminated by merging with a contaminated maximal nonseparable region. However, if these two regions merged without the searchlight configuration leaving $\bar{\Lambda}$, then they must be $\Lambda$-equivalent and therefore should be interpreted as a single contaminated maximal nonseparable region. □

Now we are ready to define the classes of schedules we are interested in.

**Definition 3.14** (Classes of Schedules)**.** *A schedule is called*

*(i)* sequential *if only one searchlight is active at a time,*

*(ii)* critical *if searchlights only rotate between critical angles, i.e., while rotating they never stop or change direction at a noncritical angle,*

*(iii)* rotation-monotone *if each searchlight is constrained to rotate either exclusively cw or exclusively ccw,*

*(iv)* contamination-monotone *if no point in the environment changes contamination state more than once, and*

*(v)* general *if it does not necessarily fall under any of the above special classes.*

We have all but stated explicitly what the exact cell decomposition of the searchlights' joint configuration space ($\mathbb{T}^N$) is. The cells are precisely the set of all closed critical rectangles of positive measure, i.e., those which are the Cartesian product of non-null critical intervals (see example Fig. 3.5). This decomposition is exact in the sense that for the Searchlight Scheduling Problem it suffices to consider only the class of schedules which travel along cell boundaries, which is the class of critical sequential schedules. This exactness will be captured rigorously in the main Theorems 3.1, 3.2, and Corollary 3.1 at the end of this section, but first we require a few more definitions and lemmas.

**Definition 3.15** (Atomic and Critical Atomic Actions)**.** *Let $\mathcal{A}^{[i]}$ denote a rotation action by a searchlight $s^{[i]}$ from an angle $\alpha_{\text{init}}$ to an angle $\alpha_{\text{fin}}$, where $\alpha_{\text{init}}$ and $\alpha_{\text{fin}}$ may or may not be critical angles. $\mathcal{A}^{[i]}$ is* atomic *if*

*(i) $s^{[i]}$ rotates monotonically cw or ccw without stopping, and*

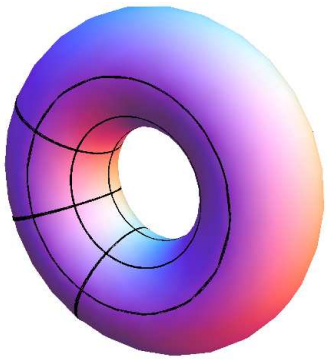


Figure 3.5. For the simple problem instance of Fig. 3.3a, each of the 2 searchlights has 3 critical angles. Together, these 6 critical angles define the exact cell decomposition of the searchlight configuration space $\mathbb{T}^2$ (shown embedded in $\mathbb{R}^3$, thick black lines are the cell boundaries).

*(ii)* $s^{[i]}$ *does not aim at any critical angle during the execution of the action except possibly* $\alpha_{\text{init}}$ *and/or* $\alpha_{\text{fin}}$.

*If* $\alpha_{\text{init}}$ *and* $\alpha_{\text{fin}}$ *are both critical angles, then* $\mathcal{A}^{[i]}$ *is called a* critical atomic action.

Simply put, a rotation action being atomic just amounts to the searchlight not changing direction of rotation nor crossing a critical angle.

**Definition 3.16** (Projections of Atomic Actions)**.** *Given an atomic action* $\mathcal{A}^{[i]}$ *there is a minimal critical interval* $\lambda^{[i]}$ *containing that action. The* forward projection *of* $\mathcal{A}^{[i]}$*, denoted* $\hat{\mathcal{A}}^{[i]}$*, rotates* $s^{[i]}$ *over all of* $\bar{\lambda}^{[i]}$ *by rotating in the same direction as* $\mathcal{A}^{[i]}$*. The* reverse projection *of* $\mathcal{A}^{[i]}$*, denoted* $\check{\mathcal{A}}^{[i]}$*, rotates* $s^{[i]}$ *over all of* $\bar{\lambda}^{[i]}$ *by rotating in the direction opposing* $\mathcal{A}^{[i]}$*. See Fig. 3.6.*

**Lemma 3.4.** *Any sequential searchlight schedule can be written as a discrete sequence of atomic rotation actions, i.e., a sequence of the form*

$$\{\mathcal{A}_m^{[i_m]}\}_{m=0}^{M-1} = \mathcal{A}_0^{[i_0]}\mathcal{A}_1^{[i_1]}\mathcal{A}_2^{[i_2]}\cdots\mathcal{A}_{M-1}^{[i_{M-1}]},$$

*where each* $\mathcal{A}_m^{[i_m]}$ *denotes an atomic rotation action by searchlight* $i_m$.

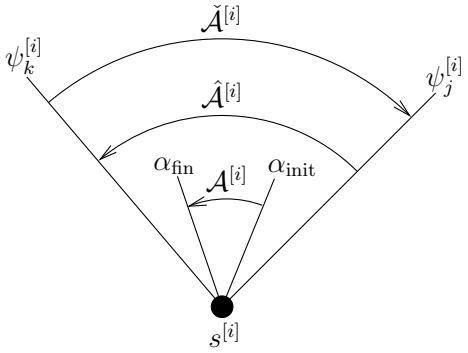



Figure 3.6. The forward projection $\hat{\mathcal{A}}^{[i]}$ and reverse projection $\check{\mathcal{A}}^{[i]}$ of an atomic action $\mathcal{A}^{[i]}$. $\psi_j^{[i]}$ and $\psi_k^{[i]}$ signify the bounding critical angles of the minimal critical interval ($\lambda^{[i]}$ in Definition 3.16) containing $\mathcal{A}^{[i]}$.

*Proof.* Given a sequential schedule represented by a sequence of actions, any non-atomic actions can be broken up into an appropriate number of atomic actions by splitting at the instances of time when a searchlight rotates over a critical angle or changes rotation direction. □

**Lemma 3.5** (Schedule Decomposition into Time Intervals)**.** *For any general search schedule $\Theta(t)$, $t \in [0, T]$, there exists a unique finite increasing sequence of times*

$$t_{0,0}, t_{0,1}, t_{0,2}, \ldots, t_{1,0}, t_{1,1}, t_{1,2}, \ldots, t_{2,0}, t_{2,1}, t_{2,2}, \ldots, t_{M,0},$$

*where*

*(i)* $t_{0,0} = 0$ *and* $t_{M,0} = T$*,*

*(ii) the times* $\{t_{i,j} |\ i \in \{1, \ldots, M-1\}$ *and* $j = 0\}$ *correspond one-to-one to the times other than* 0 *and* $T$ *when there is a change in the minimal critical rectangle containing the searchlight configuration,*[5] *and*

*(iii) the times* $\{t_{i,j} |\ i \in \{0, \ldots, M-1\}$ *and* $j > 0\}$ *correspond one-to-one to the times when one or more contaminated maximal nonseparable regions disap-*

[5] *At $t_{0,0} = 0$ and $t_{M,0} = T$ there may or may not be a change in the minimal critical rectangle containing the searchlight configuration.*

*pear but there is not concurrently a change in the minimal critical rectangle containing the searchlight configuration.*

*Proof.* Recall there are finitely many searchlights, each searchlight only has a finite number of critical angles, and searchlights may change direction of rotation only a finite number of times. From these observations, it is clear that the minimal critical rectangle containing the searchlight configuration can only change a finite number of times, i.e., $M$ is finite. Finiteness of the subsequences $t_{i,1}, t_{i,2}, t_{i,3}, \ldots$ (for $i \in \{0, 1, \ldots, M-1\}$) follows from the facts that (i) there are only finitely many maximal nonseparable regions, and (ii) while the searchlight configuration remains in the same minimal critical rectangle, any contaminated maximal nonseparable regions which disappear cannot be recontaminated (see Lemma 3.2). Uniqueness follows from the one-to-one correspondence between times and events. □

**Lemma 3.6.** *Let $\Lambda$ be the minimal critical rectangle containing some searchlight configuration. Then, modulo $\Lambda$-equivalence, the area of each maximal nonseparable region is a continuous function of the searchlight configuration when restricted to $\bar{\Lambda}$.*

*Proof.* We know from elementary geometry that the area of a polygon is a continuous function of the coordinates of its vertices. Maximal nonseparable regions are polygons whose vertices have the following property: the coordinates of the vertices are continuous functions of the searchlight configuration. □

**Lemma 3.7.** *Suppose a maximal nonseparable region $R$ can be made to disappear by a single atomic action $\mathcal{A}^{[i]}$ from a configuration $\Theta_0$ (cf Fig. 3.3). Let $\Theta_1$ be a configuration of the searchlights identical to $\Theta_0$ except that $s^{[i]}$ is aimed at an angle in the interior of the angular interval over which it sweeps according to $\mathcal{A}^{[i]}$, and let $\Lambda$ be the minimal critical rectangle containing $\Theta_1$.*[6] *Then the area of*

[6] *The important feature of $\Lambda$ is that if $\Lambda'$ is the minimal critical rectangle containing $\Theta_0$, then $\bar{\Lambda}' \subset \bar{\Lambda}$ and the searchlight configuration will remain in $\bar{\Lambda}$ while $s^{[i]}$ executes $\mathcal{A}^{[i]}$.*

*$R$ is a monotonic continuous function of each supporting searchlight's configuration (holding all other searchlights' configurations fixed) while the configuration is restricted to $\bar{\Lambda}$.*

*Proof.* Observe that $R$ cannot extend around a hole of $\mathcal{E}$, otherwise some portion of $R$ would be invisible to $s^{[i]}$. Also, there can be no searchlights located in the interior of $R$, otherwise $s^{[i]}$ would have to rotate over another searchlight to clear $R$ (which would mean $\mathcal{A}^{[i]}$ were not atomic). From these properties it follows, that while the searchlights remain in $\bar{\Lambda}$, two facts hold true: (i) the portion of $R$'s boundary formed by its supporting searchlight beams must be convex[7], and (ii) $R$ must lie entirely on one side of each supporting searchlight's beam. If any supporting beam rotates towards the interior of $R$, then $R$'s area must decrease. Likewise the area increases if any supporting beam rotates away from the interior, so we have established monotonicity. Continuity follows as in Lemma 3.6. □

**Theorem 3.1** (Reduction from General to Sequential)**.** *Any instance of the Searchlight Scheduling Problem which permits a general search schedule also permits a sequential search schedule.*

*Proof.* Let $\Theta(t)$, $t \in [0, T]$, be a general search schedule and

$$t_{0,0}, t_{0,1}, t_{0,2}, \ldots, t_{1,0}, t_{1,1}, t_{1,2}, \ldots, t_{2,0}, t_{2,1}, t_{2,2}, \ldots, t_{M,0}$$

a sequence of times as described in Lemma 3.5. We say that a schedule $\tilde{\Theta}(t)$ *emulates* another schedule $\Theta(t)$ over a time interval $[t_{\text{init}}, t_{\text{fin}}]$ if $\tilde{\Theta}(t_{\text{init}}) = \Theta(t_{\text{init}})$ and $\tilde{C}(t_{\text{init}}) = C(t_{\text{init}})$ implies $(\tilde{\Theta}(t_{\text{fin}}), \tilde{C}(t_{\text{fin}})) \preceq (\Theta(t_{\text{fin}}), C(t_{\text{fin}}))$. To prove the theorem it suffices to show a sequential schedule $\tilde{\Theta}(t)$ can be constructed which emulates $\Theta(t)$ over the time interval $[0, T]$.

We first show that $\Theta(t)$ can be emulated by a sequential schedule over each time interval of the form $[t_{i,0}, t_{i,0}+\epsilon]$, $i \in \{0, 1, \ldots, M-1\}$, where $\epsilon \in ]0, t_{i,1} - t_{i,0}[$. This is when merging can occur (review Fig. 3.3). Suppose we simply execute a

[7]By "convex" we mean that when two searchlight beams form adjacent edges of $R$, then the interior angle between those edges is less than $\pi$. The portion of $R$'s boundary formed by the environment boundary may or may not be convex.

sequence of atomic actions $\mathcal{A}_0^{[0]}\mathcal{A}_1^{[1]}\mathcal{A}_2^{[2]}\cdots\mathcal{A}_{N-1}^{[N-1]}$ which rotates each searchlight directly from $\theta^{[i]}(t_{i,0})$ to $\theta^{[i]}(t_{i,0}+\epsilon)$, $i \in \{1,\ldots,N\}$, one at a time in no particular order. It suffices to guarantee such a sequence of atomic actions does not cause any maximal nonseparable region, say $R$, to merge with a contaminated region, say $R'$, which $R$ did not merge with under $\Theta(t)$.[8] This is indeed the case, for suppose that $R$, through $\mathcal{A}_0^{[0]}\mathcal{A}_1^{[1]}\mathcal{A}_2^{[2]}\cdots\mathcal{A}_{N-1}^{[N-1]}$, does merge with such an $R'$. This is only possible if the support of $R$ changes to include a new beam that will cause the merge. The only way to change the support of $R$ in this manner is for one or more supporting beams of $R$ to cross the intersection of a combination of other searchlights' beams and $\partial\mathcal{E}$. In such a case, it must be that $R'$ just appeared as an artifact of using the sequence of atomic actions, thus $R'$ is clear (see examples in Fig. 3.7).

We next consider the existence of a sequential schedule to emulate $\Theta(t)$ only during a time interval of the form $[t_{i,0}+\epsilon, t_{i+1,0}]$, $i \in \{0,1,\ldots,M-1\}$, where $\epsilon \in ]0, t_{i,1}-t_{i,0}[$. Let $\Lambda$ be the minimal critical rectangle containing $\Theta(t_{i,0}+\epsilon)$. Recall that a maximal nonseparable region disappears if and only if its area goes to zero and this area, modulo $\Lambda$-equivalence, is a continuous function of the searchlight configuration when restricted to $\bar{\Lambda}$ (see Lemma 3.6). Together with Lemmas 3.3, this implies that all contaminated maximal nonseparable regions which are caused to disappear by $\Theta(t)$ during $(t_{i,0}, t_{i+1,0})$ can also be made to disappear one at a time by a sequential schedule. In particular, $\Theta(t)$ can be emulated over $[t_{i,0}+\epsilon, t_{i+1,0}]$ by any sequential schedule $\tilde{\Theta}(t)$ which visits successively the configurations $\Theta(t_{i,0}+\epsilon), \Theta(t_{i,1}), \Theta(t_{i,2}), \ldots, \Theta(t_{i+1,0})$ without leaving $\bar{\Lambda}$.[9]

We now know that the general schedule $\Theta(t)$ can be emulated by a sequential schedule $\tilde{\Theta}(t)$ over each interval $[t_{i,0}, t_{i+1,0}]$, $i \in \{0,1,\ldots,M-1\}$, such that $\tilde{\Theta}(t_{i+1,0}) = \Theta(t_{i+1,0})$. Therefore concatenating the sequential schedules produces a sequential search schedule $\tilde{\Theta}(t)$ emulating $\Theta(t)$ over the duration $[0,T]$. □

[8] If multiple regions merge into one under $\Theta(t)$, then these regions may instead merge one at a time when using $\mathcal{A}_0^{[0]}\mathcal{A}_1^{[1]}\mathcal{A}_2^{[2]}\cdots\mathcal{A}_{N-1}^{[N-1]}$, but this does not affect our line of argument.

[9] In our definition of emulation, it is only important to visit $\Theta(t_{i,0}+\epsilon)$ first and $\Theta(t_{i+1,0})$ last, otherwise the order of visiting the configurations does not matter.

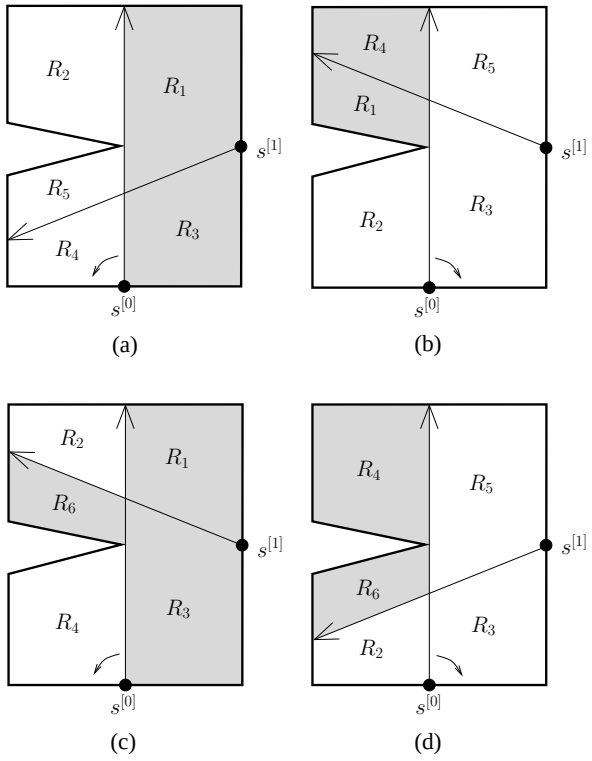



Figure 3.7. Gray regions are clear. The effect of a searchlight $s^{[0]}$ executing the action indicated by the smaller arrow is (a) $R_1$ merges with $R_2$, (b) $R_1$ merges with $R_2$, (c) $R_1$ merges with $R_2$ and $R_3$ merges with $R_6$, (d) $R_4$ merges with $R_6$. (a) and (b) are analogous to (a) and (b), resp., of Fig. 3.3. On the other hand, (c) and (d) show the effects of $s^{[1]}$ rotating over the intersection of $s^{[0]}$'s beam with the reflex vertex, before $s^{[0]}$'s action is executed as in (a) and (b), respectively. Note that $R_6$ in (c) and (d) are clear because they are simply artifacts which appeared as a result of $s^{[1]}$ rotating over the intersection of $s^{[0]}$'s beam with the reflex vertex.

**Theorem 3.2** (Reduction from Sequential to Critical Sequential)**.** *Any instance of the Searchlight Scheduling Problem which permits a sequential search schedule also permits a critical sequential search schedule.*

*Proof.* By Lemma 3.4 it suffices to show that given an atomic sequential schedule, i.e., a search schedule written as a sequence of atomic actions $\{\mathcal{A}_m^{[i_m]}\}_{m=0}^{M-1} =$

$\mathcal{A}_0^{[i_0]}\mathcal{A}_1^{[i_1]}\mathcal{A}_2^{[i_2]}\cdots\mathcal{A}_{M-1}^{[i_{M-1}]}$, we can construct another search schedule expressed as a sequence of critical atomic actions. Reducing the problem further, suppose that from an atomic sequential search schedule $\{\mathcal{A}_m^{[i_m]}\}_{m=0}^{M-1}$ we are able to construct a new atomic sequential search schedule $\{\tilde{\mathcal{A}}_m^{[i_m]}\}_{m=0}^{\tilde{M}-1}$ such that for an arbitrary searchlight, say $s^{[0]}$,

(i) the actions executed by searchlights other than $s^{[0]}$ are unaltered, and

(ii) the actions executed by $s^{[0]}$ are exclusively critical atomic (though they may increase in number).

If we can find a procedure to construct such a schedule, then this procedure could be repeated for each $s^{[i]}$, $i \in \{0,\ldots,N-1\}$, until we are left with a critical atomic sequential schedule. We show such a procedure exists.

Let $(\Theta_m, C_m)$ denote the information state of the original schedule $\{\mathcal{A}_m^{[i_m]}\}_{m=0}^{M-1}$ just before the $m$th action is executed. Without loss of generality, assume that in $\{\mathcal{A}_m^{[i_m]}\}_{m=0}^{M-1}$ all searchlights are initially aimed at critical angles. We construct from $\{\mathcal{A}_m^{[i_m]}\}_{m=0}^{M-1}$ another search schedule $\{\tilde{\mathcal{A}}_m^{[i_m]}\}_{m=0}^{\tilde{M}-1}$ as described above essentially by modifying $\{\mathcal{A}_m^{[i_m]}\}_{m=0}^{M-1}$ action by action. Suppose $\mathcal{A}_k^{[0]}$ is the first action which rotates $s^{[0]}$ from a critical angle $\psi_1^{[0]}$ to a noncritical angle $\alpha$ contained in the critical interval $[\psi_1^{[0]}, \psi_2^{[0]}]$. We can let $\tilde{\mathcal{A}}_m^{[i_m]} = \mathcal{A}_m^{[i_m]}$, $m \in \{0,\ldots,k-1\}$, but we do not want $s^{[0]}$ to stop at a noncritical angle, so we set $\tilde{\mathcal{A}}_k^{[0]} = \hat{\mathcal{A}}_k^{[0]}$ (recall Definition 3.16). Since $\tilde{\mathcal{A}}_k^{[0]}$ sweeps over a larger region than $\mathcal{A}_k^{[0]}$ and does not cause any merging which $\mathcal{A}_k^{[0]}$ does not, it must clear the same maximal nonseparable regions as $\mathcal{A}_k^{[0]}$. We only need to worry about how this change in $s^{[0]}$'s configuration will effect subsequent actions by other searchlights. Suppose the next action is $\mathcal{A}_{k+1}^{[1]}$. We keep $\tilde{\mathcal{A}}_{k+1}^{[1]} = \mathcal{A}_{k+1}^{[1]}$, but since during $\tilde{\mathcal{A}}_{k+1}^{[1]}$ $s^{[0]}$ was aimed at $\psi_2^{[0]}$ (instead of $\alpha$), the effect of $\tilde{\mathcal{A}}_{k+1}^{[1]}$ may be different than $\mathcal{A}_{k+1}^{[1]}$. In particular, $\tilde{\mathcal{A}}_{k+1}^{[1]}$ may not clear all the same maximal nonseparable regions which $\mathcal{A}_{k+1}^{[1]}$ did. The monotonicity property of Lemma 3.7 guarantees we can compensate for this difference simply by choosing $\tilde{\mathcal{A}}_{k+2}^{[0]} = \check{\mathcal{A}}_k^{[0]}$. The important result of executing such a $\tilde{\mathcal{A}}_{k+2}^{[0]}$ is that if we were to then rotate $s^{[0]}$ from $\psi_1^{[0]}$ directly back to $\alpha$ (though we do not actually

do this because we only want $s^{[0]}$ to stop at critical angles), then we would end up in the configuration $\Theta_{k+2}$ with contamination state $C \preceq C_{k+2}$. So far we have constructed a schedule $\{\tilde{\mathcal{A}}_m^{[i_m]}\}_{m=0}^{k+2}$ from $\{\mathcal{A}_m^{[i_m]}\}_{m=0}^{k+1}$. The procedure continues alternately taking an action from $\{\mathcal{A}_m^{[i_m]}\}_{m=k+2}^{M}$ and rotating $s^{[0]}$ over the critical interval $[\psi_1^{[0]}, \psi_2^{[0]}]$. After every pair of such actions, the monotonicity property of Lemma 3.7 guarantees the same maximal nonseparable regions will be cleared as in the original schedule. In the end we are left with a sequential search schedule $\{\tilde{\mathcal{A}}_m^{[i_m]}\}_{m=0}^{\tilde{M}-1}$ which satisfies the above enumerated requirements. □

Putting together Theorems 3.1 and 3.2, we finally arrive at the main solution space reduction result.

**Corollary 3.1** (Main Reduction Result)**.** *Any instance of the Searchlight Scheduling Problem which permits a general search schedule also permits a critical sequential search schedule.*

*Proof.* Immediate from combining Theorems 3.1 and 3.2. □

An interesting and open problem is whether it is possible to reduce the solution space even further, e.g., as in Conjecture 3.1 below. Further reduction of the solution space could allow for faster computation of search schedules.

**Conjecture 3.1** (Rotation- and Contamination-Monotonicity)**.** *Any instance of the Searchlight Scheduling Problem which permits a general search schedule also permits a rotation- and contamination-monotone critical sequential search schedule.*[10]

[10] *It is intended that there would exist a rotation- and contamination-monotone critical sequential search schedule from some initial condition, not necessarily from any initial condition.*

## 3.4 A Complete Algorithm

The solution space reduction result Corollary 3.1 tells us that if we can systematically search the space of critical sequential schedules, then we are guaranteed to find a search schedule if one exists. Our algorithm does just this by searching for a solution trajectory through a discretization of the information space. Every critical sequential search schedule, by definition, can be represented by a sequence of critical atomic actions connecting critical information states (as in, e.g., Fig. 3.11), thus the information space discretization is defined as follows.

**Definition 3.17** (Directed Information Graph $\mathcal{G}_{\mathcal{I}}$)**.**

*(i) nodes correspond to critical information states, and*

*(ii) there is a directed edge from one node, say $x$, to another node, say $x'$, if and only if it is possible to reach $x'$ from $x$ by executing a single critical atomic action.*

If a search schedule exists, then our algorithm will find it by searching $\mathcal{G}_{\mathcal{I}}$ for a path from a completely contaminated node to a completely clear node. In its entirety the algorithm consists of two parts: (i) geometric preprocessing which extracts combinatorial information from the problem instance geometry, followed by (ii) systematic search of the information graph. We detail these parts separately in Sections 3.4.1 and 3.4.2, then provide a discussion of implementation and computed examples in Section 3.4.3.

### 3.4.1 Geometric Preprocessing

The geometric preprocessing, taking the environment geometry and searchlight positions as input, computes an environment partition and a graph by

Table 3.1. Geometric Preprocessing

| **Input:** geometric problem instance $(\mathcal{E}, P)$ |
|---|
| 1: **for all** searchlights $i = 1, \ldots, N$ **do** |
| 2: compute visibility polygon $\mathcal{V}(p^{[i]})$; |
| 3: extract critical angles $\Psi^{[i]} = \{\psi_1^{[i]}, \psi_2^{[i]}, \ldots, \psi_{n_\psi^{[i]}}^{[i]}\}$ from $\mathcal{V}(p^{[i]})$; |
| 4: compute cells $\gamma_1, \gamma_2, \ldots, \gamma_{n_\gamma}$ of environment partition $\Gamma$; |
| 5: compute partition dual graph $\mathcal{G}_\Gamma$; |

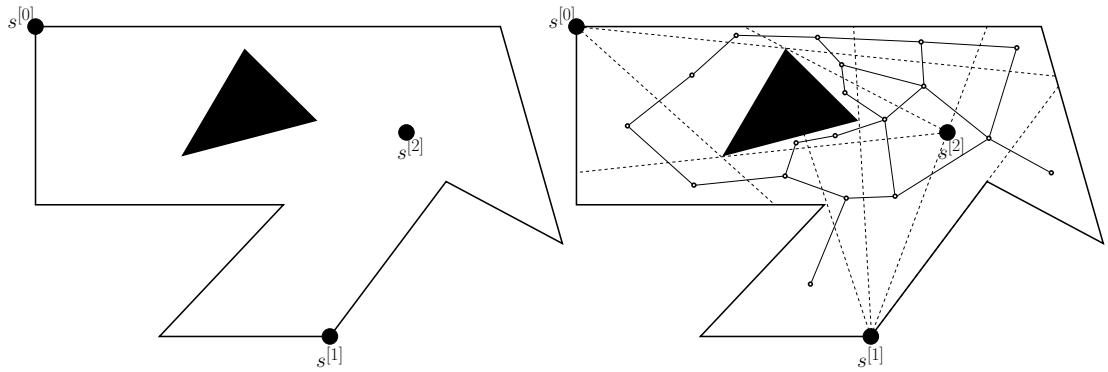



Figure 3.8. The geometric preprocessing part of the complete algorithm (Table 3.1, Section 3.4) is illustrated using a simple problem instance having three searchlights and one hole. First a geometric description of the problem instance is taken as input (left). This consists of the environment geometry $\mathcal{E}$ together with the searchlight locations $P$. Next (right), the critical angles $\Psi$ of the searchlights (dashed lines) and environment partition $\Gamma$ are computed. Each cell of $\Gamma$ is either completely clear or completely contaminated on any node of the information graph $\mathcal{G}_\mathcal{I}$. Finally, the dual graph $\mathcal{G}_\Gamma$ of the environment partition is computed. This instance was solved by our C++ implementation of the complete algorithm. The computed solution is illustrated in Fig. 3.11, statistics in Table 3.3.

means of the sequence of computations shown in Table 3.1. Each of the searchlights' visibility polygons $\mathcal{V}(p^{[i]})$, $i \in \{1, \ldots, N\}$, will have at most $n$ edges and can be computed in $\mathcal{O}(n \log n)$ time [74]. Let $\Psi = \{\Psi^{[1]}, \Psi^{[2]}, \ldots, \Psi^{[N]}\}$, where

$\Psi^{[i]} = \{\psi_1^{[i]}, \psi_2^{[i]}, \ldots, \psi_{n_\psi^{[i]}}^{[i]}\}$ denotes a list of the $i$th searchlight's critical angles. Using Definition 3.8, each $\Psi^{[i]}$ can easily be extracted from $\mathcal{V}(p^{[i]})$ by checking for (i) radially aligned edges of $\mathcal{V}(p^{[i]})$ in $\mathcal{O}(n)$ time, and (ii) inclusion of other searchlights in $\mathcal{V}(p^{[i]})$ in $\mathcal{O}(Nn)$ time (point-in-polygon test) [88]. Adding these time complexities gives a bound on the total time to compute $\Psi$.

**Lemma 3.8.** *The critical angles $\Psi$ can be computed in $\mathcal{O}(Nn \log n + N^2 n)$ time.*

Lemma 3.9 gives an upper bound on the size of each $\Psi^{[i]}$.

**Lemma 3.9.** *A searchlight can have no more than $r+2N$ critical angles if located on $\partial\mathcal{E}$, otherwise no more than $r + 2N - 2$ if located in $\mathcal{E}^\circ$.*

*Proof.* These bounds follow directly from the Definition 3.8 of critical angles. A searchlight located on $\partial\mathcal{E}$ may have 2 critical angles due to adjacent edges of $\partial\mathcal{E}$, $2(N-1)$ due to line-of-sight with other searchlights, and $r$ due to reflex vertices of $\mathcal{E}$, hence $2 + 2(N-1) + r = r + 2N$. For a searchlight located in $\mathcal{E}^\circ$ the only difference is that there can be no critical angles due to adjacent edges of $\partial\mathcal{E}$. □

In most instances, however, the number of critical angles is far less than the bound in Lemma 3.9.

We now describe how the environment partition $\Gamma$ is constructed. For each searchlight $s^{[i]}$, $i \in \{1, \ldots, N\}$, and critical angle $\psi_j^{[i]}$, $j \in \{1, \ldots, n_\psi^{[i]}\}$, the $j$th *critical segment* of the $i$th searchlight is the closed line segment consisting of all points illuminated by $s^{[i]}$ when aimed in the direction $\psi_j^{[i]}$. The critical segments together partition $\mathcal{E}$ into a finite set of simply connected polygonal *cells*.[11] Examples are shown by the dashed lines in Fig. 3.8, 3.9, 3.10. Representing $\mathcal{E}$ as a

[11] Recall that in Section 3.3 we spoke of cells as part of an exact cell decomposition of the searchlight configuration space. These cells on $\mathbb{T}^N$ were a theoretical tool for obtaining the main reduction result Corollary 3.1. Since as part of the geometric preprocessing we partition the environment $\mathcal{E}$ into polygonal cells, we stipulate now that, whenever the term "cells" appears in Section 3.4, it is always the environment partition cells which are intended, not the cells of the exact cell decomposition of $\mathbb{T}^N$.

list of cells $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_{n_\gamma}\}$ allows one to encode the contamination state by a binary $n_\gamma$-tuple $C_\Gamma$. In $C_\Gamma$, each cell is labeled either "0" for clear or "1" for contaminated. This representation of a contamination state, together with the searchlights' joint configuration, is used to represent a node of $\mathcal{G}_\mathcal{I}$. Lemma 3.10 gives an upper bound on the time to compute $\Gamma$.

**Lemma 3.10.** *The cells $\gamma_1, \gamma_2, \ldots, \gamma_{n_\gamma}$ of the environment partition $\Gamma$ can be computed in $\mathcal{O}((n + Nr + 2N^2)^8)$ time.*

*Proof.* Constructing the cells of $\Gamma$ amounts to computing the faces of an arrangement of line segments. The faces of an arrangement of $L$ line segments can be computed in $\mathcal{O}(L^8)$ time. We count the line segments contributing to our arrangement. There are $n$ segments due to $\mathcal{E}$, and from Lemma 3.9 at most $N(r + 2N)$ critical segments. Substituting $L = n + Nr + 2N^2$, we find the cells can be computed in $\mathcal{O}((n + Nr + 2N^2)^8)$ time. Because arrangements constitute an already well studied area of computational geometry, and for the sake of brevity, we omit further detail.[89] □

Lemma 3.11 gives an upper bound on the number of cells in $\Gamma$.

**Lemma 3.11.** *The number $n_\Gamma$ of cells in the environment partition $\Gamma$ is $\mathcal{O}(N^4 + r^2N^2)$.*

*Proof.* Given and arrangement of $L \in \mathbb{N}$ lines in the plane, the maximum number of regions they partition the plane into is $\frac{L(L+1)}{2} + 1$; see [89]. According to Lemma 3.9, the interior of $\mathcal{E}$ is partitioned by at most $N(r + 2N) = rN + 2N^2$ critical segments. Setting $L = rN + 2N^2$, we see that the affine hulls of the critical segments together partition the plane into at most

$$\frac{(rN + 2N^2)(rN + 2N^2 + 1)}{2} + 1 = \frac{r^2N^2 + 4rN^3 + rN + 4N^4 + 2N^2}{2} + 1$$

regions. This gives a conservative upper bound on the order of $n_\Gamma$. We did not have to take into account the $n$ line segments of $\mathcal{E}$ because only the $r$ reflex vertices

can cause the cell count to increase (by one each), so the order would not have been affected. □

The final task of geometric preprocessing is to compute the undirected dual graph of $\Gamma$.

**Definition 3.18** (Dual Partition Graph)**.** *The dual graph $\mathcal{G}_\Gamma$ of $\Gamma$ is the undirected graph defined as follows:*

*(i) its nodes are the polygonal cells $\{\gamma_1, \gamma_2, \ldots, \gamma_{n_\gamma}\}$ of $\Gamma$, and*

*(ii) there is a (undirected) edge from one node, say $\gamma$, to another node, say $\gamma'$, if and only if the polygons $\gamma$ and $\gamma'$ share an edge.*

An example of a partition dual graph is shown in Fig. 3.8. Encoding the adjacency information of the environment partition cells, $\mathcal{G}_\Gamma$ will later allow us to compute the recontamination which can occur during an information state transition. In this way, $\mathcal{G}_\Gamma$ is a parameter of the information state transition function $f_{\mathcal{G}_\Gamma}(x, u)$ as we will see in Section 3.4.2 and Table 3.2. Lemma 3.12 gives an upper bound on the time to compute $\mathcal{G}_\Gamma$.

**Lemma 3.12.** *The dual graph $\mathcal{G}_\Gamma$ of the evironment partition $\Gamma$ can be computed in $\mathcal{O}((N^4 + r^2N^2)^2(n + Nr + N^2)^2)$ time.*

*Proof.* $\mathcal{G}_\Gamma$ can be easily computed by comparing every edge of every cell. When an edge of two cells matches, then an edge is added between the respective nodes of $\mathcal{G}_\Gamma$. Lemma 3.11 tells us we must check all pairs of $\mathcal{O}(N^4 + r^2N^2)$ cells. Each cell has at most $n + Nr + 2N^2$ edges, so for each of the $\mathcal{O}((N^4 + r^2N^2)^2)$ pairs of cells, $\mathcal{O}((n + Nr + N^2)^2)$ edges must be compared. The total time complexity is therefore $\mathcal{O}((N^4 + r^2N^2)^2(n + Nr + N^2)^2)$. □

Together Lemmas 3.8, 3.10, and 3.12 tell us that the total time complexity of geometric preprocessing is polynomial in the problem instance parameters $N$,

$n$, and $r$. It may be possible to compute the environment partition $\Gamma$ and its dual graph $\mathcal{G}_\Gamma$ faster than our bounds suggest. However, we have not spent much effort optimizing the geometric preprocessing because, as indicated by computed examples (see, e.g., Table 3.3), the overall time complexity of our algorithm is dominated by the information graph search described in Section 3.4.2.

### 3.4.2 Searching the Information Graph $\mathcal{G}_\mathcal{I}$

The information graph $\mathcal{G}_\mathcal{I}$ can be searched using any systematic graph search algorithm such as breadth-first, Dijkstra, or $A^*$. Graph search algorithms are surveyed nicely in, e.g., [62] and [90]. We use breadth-first for simplicity. A pseudocode is provided in Table 3.2, where our notation closely follows that used on page 33 of [62]. Since $\mathcal{G}_\mathcal{I}$ is typically very large, containing many irrelevant and unreachable nodes, we do not precompute $\mathcal{G}_\mathcal{I}$, but instead nodes are added to the representation only as visited by the graph search. The search begins by pushing an initial information state $x_0$ onto the FIFO (First-In First-Out) queue $Q$. In $x_0$, the contamination state $C_\Gamma = (1, 1, 1 \ldots, 1)$ (environment completely contaminated) and the searchlight configuration may be chosen arbitrarily. At each iteration of the main loop a node $x$ is popped off the queue $Q$, added to the search tree $T$, and its out-neighbors (in $\mathcal{G}_\mathcal{I}$) are computed. If an out-neighbor is a goal node, i.e., an information state having a completely clear environment ($C_\Gamma = (0, 0, 0, \ldots, 0)$), then the algorithm returns SUCCESS. If an out-neighbor is not a goal node and it is not redundant (not already in $Q$ or $T$), then it is added onto the queue. FAILURE is returned if every node of $\mathcal{G}_\mathcal{I}$ was visited and no goal node found, which means no solution exists. When the algorithm does return SUCCESS, a critical sequential search schedule can be recovered by

backtracing pointers through the search tree and storing the sequence of critical atomic actions.

Theorem 3.3 and Corollary 3.2 provide upper bounds on the size of $\mathcal{G}_{\mathcal{I}}$.

**Theorem 3.3** (Number of Nodes in $\mathcal{G}_{\mathcal{I}}$)**.** *The number of nodes in the information graph $\mathcal{G}_{\mathcal{I}}$ is*[12] $(2N + r)^N 2^{\mathcal{O}(N^4+N^2r^2)})$.

*Proof.* Follows from Lemmas 3.11 and 3.9. To count the number of unique contamination labelings we raise 2 to the bound on the number of cells. $(2N+r)^N$ is an upper bound on the number of unique critical searchlight configurations. □

Stirling's approximation says that $N^N \approx N!$, so the upper bound in Theorem 3.3 grows with $N$ worse than $N!$. The following Corollary 3.2 of Theorem 3.3 shows, however, that if there is an upper bound $k$ on the maximum number of critical angles any single searchlight has, then the number of nodes in $\mathcal{G}_{\mathcal{I}}$ can be bounded by a function which is only exponential in $N$.

**Corollary 3.2** (Number of Nodes in $\mathcal{G}_{\mathcal{I}}$)**.** *Suppose that each searchlight in an instance has at most k critical angles. Then the number of nodes in the information graph $\mathcal{G}_{\mathcal{I}}$ is $k^N 2^{\mathcal{O}(N^2k^2+r)}$.*

*Proof.* The number of line segments forming the environment partition can be no greater than $Nk + n$, so setting $\xi = Nk + n$ in the formula in the proof of Lemma 3.11 and adding $r$ gives an upper bound on the number of cells. To count the number of unique (binary) contamination labelings, we raise 2 to the bound on the number of cells. $k^N$ is an upper bound on the number of unique critical searchlight configurations. The resulting upper bound on the number of nodes in $\mathcal{G}_{\mathcal{I}}$ is $k^N 2^{\frac{1}{2}N^2k^2+\frac{1}{2}Nk+r+1}$. □

[12] *A function $g(x)$ is in the set $2^{\mathcal{O}(h(x))}$ if $\exists\ x_0$ and $c \in \mathbb{R}_{>0}$ such that $x > x_0 \implies g(x) < 2^{ch(x)}$. Note that $\mathcal{O}(2^{h(x)})$ is a proper subset of $2^{\mathcal{O}(h(x))}$.*

Table 3.2. Breadth First Search of Information Graph $\mathcal{G}_{\mathcal{I}}$

| | | |
|---|---|---|
| $x_{\text{initial}}$ | := | initial $\mathcal{G}_{\mathcal{I}}$ node with $C_\Gamma = (1, 1, 1, \ldots, 1)$ |
| $X_{\text{clear}}$ | := | set of $\mathcal{G}_{\mathcal{I}}$ nodes with $C_\Gamma = (0, 0, 0, \ldots, 0)$ |
| $Q$ | := | FIFO queue of alive $\mathcal{G}_{\mathcal{I}}$ nodes |
| $T$ | := | search tree of dead $\mathcal{G}_{\mathcal{I}}$ nodes |
| $U$ | := | set of critical atomic actions |
| $f_{\mathcal{G}_\Gamma}(x, u)$ | := | information state transition function |

```
 1: Q.Insert(x_initial);
 2: while Q not empty do
 3:    x ← Q.PopFirst();
 4:    T.Insert(x);
 5:    if x ∈ X_clear then
 6:       return SUCCESS;
 7:    for all u ∈ U do
 8:       x' ← f_{G_Γ}(x, u);
 9:       if x' ∉ Q and x' ∉ T then
10:          Q.Insert(x');
11: return FAILURE;
```

The bounds on the size of $\mathcal{G}_\mathcal{I}$ given in Theorem 3.3 Corollary 3.2 could be used to derive a bound on the time complexity of the graph search, however we do not do this because we believe such bounds would be too loose. Instead, in Theorem 3.4 we derive a bound which reflects the output sensitivity of the computation time.

**Theorem 3.4** (Time Complexity of $\mathcal{G}_\mathcal{I}$ Breadth-First Search)**.** *Suppose that, for a particular problem instance $(\mathcal{E}, P)$ and initial information state $x_0$, there exists a search schedule consisting of $M^*$ critical atomic actions, and $M^*$ is the smallest such number. Then performing the breadth-first search of $\mathcal{G}_\mathcal{I}$ shown in Table 3.2 starting from $x_0$ takes time $\mathcal{O}((N^4 + r^2N^2)(2N)^{2M^*}))$.*

*Proof.* Referring to Table 3.2, observe that the number of possible critical atomic actions $|U| = 2N$ (each searchlight can rotate either cw or ccw). This means each node of $\mathcal{G}_\mathcal{I}$ has at most $2N$ out-neighbors. Taking $2N$ as the branching factor of the breadth-first search tree, and knowing the search will terminate at depth $M^*$, we see that $\mathcal{O}((2N)^{M^*})$ nodes will have been visited. To generate each node (except for $x_0$) an information state transition must be computed, which can be done in $\mathcal{O}(n_\Gamma)$ time using a technique called *floodfill*.[13] So far the total time complexity we have accounted for is $\mathcal{O}(n_\Gamma(2N)^{M^*})$. Additional complexity is added by each node being compared with every other node to avoid redundancy in the search tree. To check whether two information states are equal costs $\mathcal{O}(N + n_\Gamma)$ time because the searchlight configurations and contamination state of each cell must be compared. There are $\mathcal{O}((2N)^{2M^*})$ pairs of nodes, so the redundancy checks result in total time complexity $\mathcal{O}((N + n_\Gamma)(2N)^{2M^*})$. Using Lemma 3.11 to substitute $\mathcal{O}(n_\Gamma) = \mathcal{O}(N^4+r^2N^2)$, we obtain total time complexity $\mathcal{O}((N^4 + r^2N^2)(2N)^{2M^*})$. □

[13] *Floodfill* is technique commonly used in computer graphics for painting connected regions of rasterized images.

### 3.4.3 Implementation and Computed Examples

We have implemented our algorithm, both geometric preprocessing and information graph breadth-first search, in C++ on a 2.33GHz i686 processor using the Standard Library and the VisiLibity library for visibility computations.[81] The VisiLibity Library, which we used only for the geometric preprocessing step, uses so-called $\epsilon$-geometry for robustness ([91], [92]). Table 3.3 shows statistics from the computed examples of Fig. 3.1, 3.8, 3.9, and 3.10. Fig. 3.11 shows a graphical illustration of the critical sequential search schedule computed for the instance in Fig. 3.8.

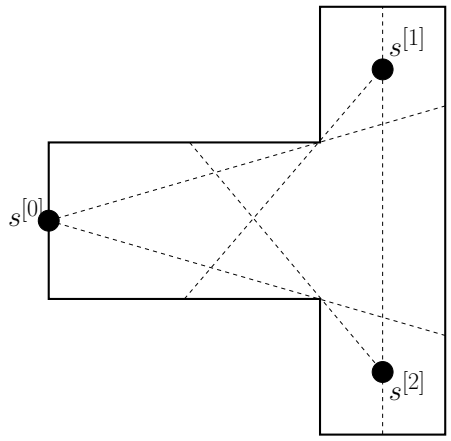



Figure 3.9. Dashed lines show the critical angles for each searchlight and also partition the environment into discrete simply connected polygonal cells. Each cell is either completely clear or completely contaminated on any node of the information graph. This instance was solved by our C++ implementation of the complete algorithm described in Section 3.4. Computation statistics are found in Table 3.3.

We have tested dozens of problem instances and although the algorithm works well for instances with up to 4 guards and 5 critical angles per guard (such as the examples shown in Fig. 3.1, 3.8, 3.9, 3.10), it seems there is a very rapid

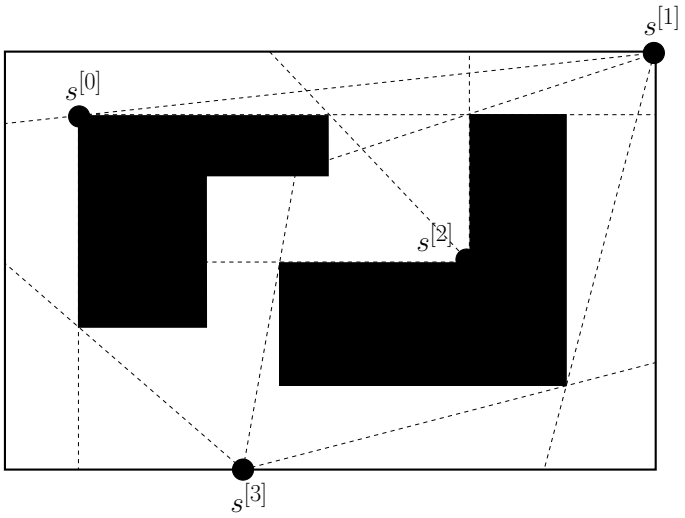



Figure 3.10. Dashed lines show the critical angles for each searchlight and also partition the environment into discrete simply connected polygonal cells. Each cell is either completely clear or completely contaminated on any node of the information graph. This instance was solved by our C++ implementation of the complete algorithm described in Section 3.4. Computation statistics are found in Table 3.3.

combinatorial explosion for problem instances even slightly more complex. For example, one problem instance we tested had $n = 21$ vertices, $r = 14$ reflex vertices, $h = 3$ holes, $N = 5$ searchlights, and it computed for over 13 hours without finding a solution. This raises an important open question which we do not answer in this chapter, namely whether the general Searchlight Scheduling Problem is NP-hard.

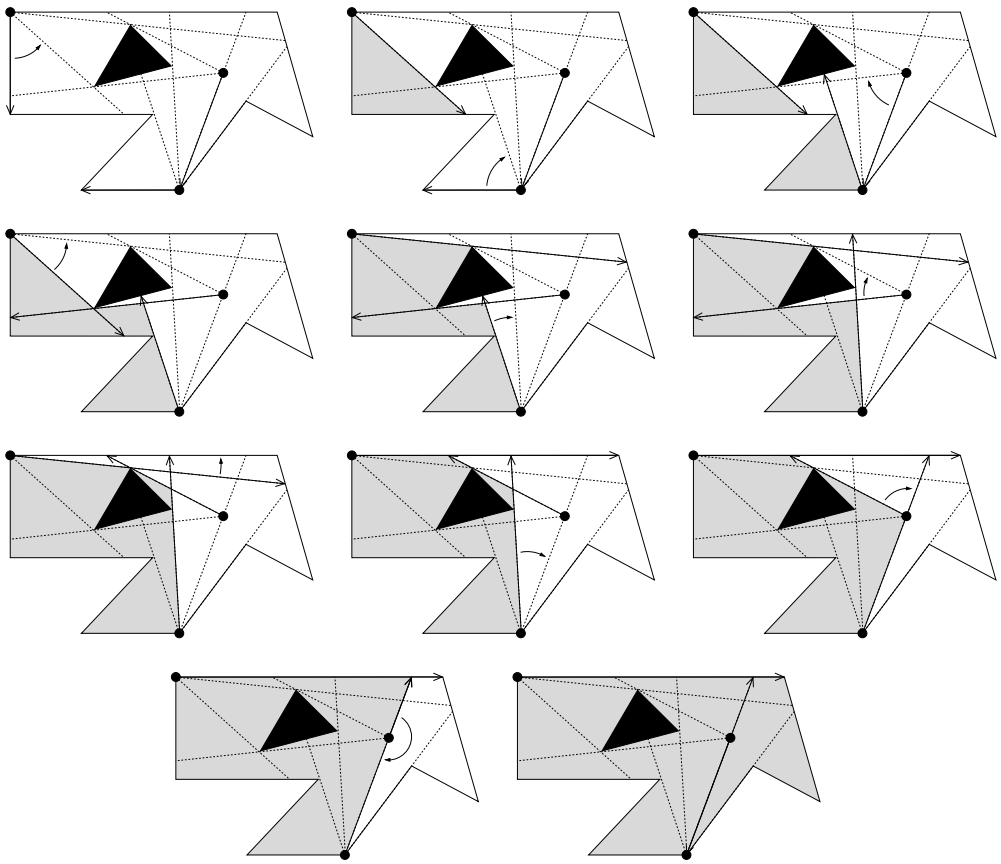


Figure 3.11. From left to right, top to bottom, here is shown a critical sequential search schedule computed by our C++ implementation of the complete algorithm described in Section 3.4 (same instance as Fig. 3.8). Grey regions are clear, the smaller arrows indicate which searchlight will execute a critical atomic action at each step of the sequence. Computation statistics are found in Table 3.3.

Table 3.3. Statistics from computed examples.

| Problem Instance | Guards | Edges in Environment | Cells in Environment Partition | Information Graph Nodes Visited | Geometric Preprocessing Time (seconds) | Information Graph Search Time (seconds) | Total Computation Time (seconds) |
|---|---|---|---|---|---|---|---|
| Fig. 3.1 | 2 | 8 | 6 | 13 | $< 0.01$ | $< 0.01$ | $< 0.01$ |
| Fig. 3.8, 3.11 | 3 | 11 | 19 | 497 | 0.03 | 0.02 | 0.05 |
| Fig. 3.9 | 3 | 8 | 15 | 464 | 0.02 | 0.01 | 0.03 |
| Fig. 3.10 | 4 | 16 | 22 | 5401 | 0.05 | 1.79 | 1.84 |

## 3.5 Extension to Searchlights with Finite Field of View

A searchlight senses only along a ray, but many real sensors, such as security cameras, have a finite field of view. This motivates the definition of *$\phi$-searchlight*,[14] which is identical to a searchlight except instead of sensing only along a ray, it can sense anywhere within a finite field of view measured by an angle $\phi$ as illustrated in Fig. 3.12. Now we can define the *$\phi$-Searchlight Scheduling Problem*:

> Given $N$ $\phi$-searchlights with finite fields of view $\phi^{[0]}, \phi^{[1]}, \phi^{[2]}, \ldots, \phi^{[N]}$, find a rotation schedule such that any target in an environment will necessarily be detected in finite time.

One must ask whether there is actually any advantage to having searchlights with finite field of view, e.g., whether there are problem instances which can be

[14]The name "$\phi$-searchlight" was inspired by the "$\phi$-searchers" of [27], the difference being that a "$\phi$-searcher" can rotate and translate, whereas a "$\phi$-searchlight" can only rotate.

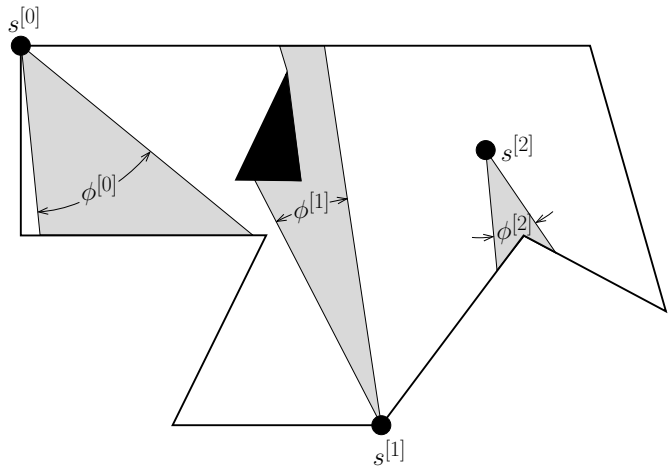



Figure 3.12. In the $\phi$-Searchlight Scheduling Problem, each searchlight may have a different field of view $\phi^{[0]}, \phi^{[1]}, \phi^{[2]}, \ldots, \phi^{[N]}$.

solved with $\phi$-searchlights but not by searchlights. Indeed there are instances where greater fields of view enable a solution.

**Lemma 3.13** (Increased Field of View Advantage)**.** *As the field of view of $\phi$-searchlights increases, the set of solvable problem instances grows.*

*Proof.* In the simple "hour-glass"[15] example of Fig. 3.13, there is no solution unless both searchlights have a field of view $\phi_{\min}$ or greater. □

The algorithm we have described in Section 3.4 could be used with $\phi$-searchlights as is, but this would not take advantage of the added capabilities offered by greater fields of view. In fact only a small modification is necessary to obtain a complete algorithm for $\phi$-searchlight scheduling. We need only redefine the critical angles. Observe that for a $\phi$-searchlight, critical visibility events can only occur when there is a change in the set of searchlight critical angles (as per Definition 3.8) illuminated by its field of view. Therefore, taking the configuration $\theta^{[i]}$ of a $\phi$-

[15]Thanks to Nicola Ceccarelli for suggesting this example.

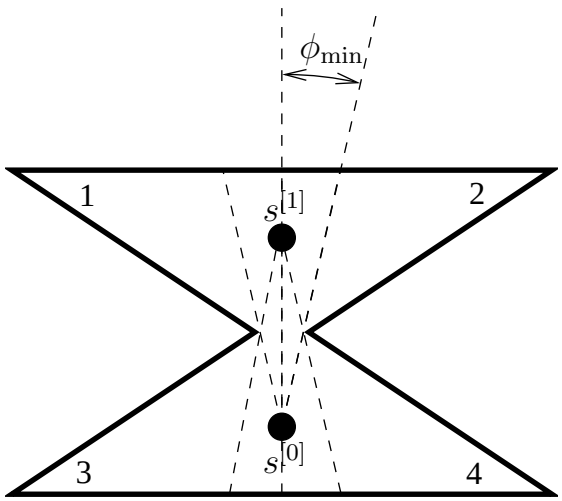



Figure 3.13. In this simple "hour-glass" example, the environment cannot be cleared unless both $\phi$-searchlights have field of view at least $\phi_{\min}$. To see why, imagine the fields of view are both less than $\phi_{\min}$ and notice if you clear any one of the corners (labeled 1, 2, 3, and 4), then trying to clear a second corner cannot be accomplished without contaminating the first. On the other hand, if both fields of view are $\phi_{\min}$ or greater, the problem is easily solvable because each $\phi$-searchlight is able to simultaneously illuminate one of the corners and the other $\phi-$searchlight

searchlight $s^{[i]}$ to be the angular position of the bisector of its field of view,[16] we have the following definition.

**Definition 3.19** ($\phi$-Searchlight Critical Angle)**.** *An angle $\psi$ is a* critical angle *of a $\phi$-searchlight $s^{[i]}$ if $\theta^{[i]}$ passing through $\psi$ implies a change in the set of searchlight critical angles illuminated by $s^{[i]}$'s field of view.*

Analogous to Definition 3.14, we can define a critical sequential schedule for $\phi$-searchlights to be one in which only one $\phi$-searchlight is active at a time and each $\phi$-searchlight may only rotate between $\phi$-searchlight critical angles. Furthermore,

[16]Taking the bisector is rather arbitrary, we just need a reference angle.

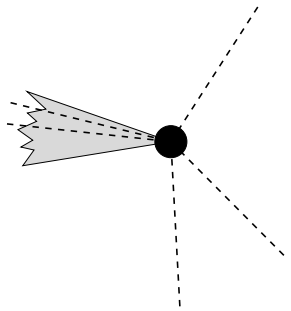


Figure 3.14. Supposing the dotted lines are searchlight critical angles, here is shown that a $\phi$-searchlight may have multiple searchlight critical angles in its view at any given time. The $\phi$-searchlight critical angles are thus defined in terms of the searchlight critical angles as in Definition 3.19.

all the Lemmas, Theorems, and Corollaries which we proved for searchlights in Section 3.3 can be proven analogously for $\phi$-searchlights simply by substituting the new definition of critical angle. The proofs are so similar that we omit them and simply state the main (solution space) reduction result.

**Corollary 3.3.** *Any instance of the $\phi$-Searchlight Scheduling Problem which permits a general search schedule also permits a critical sequential search schedule.*

Just as Corollary 3.1 led to the design of the complete algorithm for searchlight scheduling in Section 3.4, Corollary 3.3 allows a complete algorithm for $\phi$-searchlight scheduling.

## 3.6 Conclusions

In this chapter we have shown that the Searchlight Scheduling Problem can be reduced to a path planning problem through an appropriate information graph. The proof was based on an exact cell decomposition of the searchlights' toroidal

configuration space. Using the reduction result we designed a complete algorithm for searchlight scheduling. The algorithm is divided into two parts. First, geometric preprocessing is performed in time polynomial in the number of guards and environment vertices. Second, the information graph is searched breadth-first. Our time complexity upper bound for the information graph breadth-first search is exponential in the output size. Although it remains an important open question whether the general Searchlight Scheduling Problem is NP-hard, computed examples demonstrated that the algorithm can be practical for problem instances of useful size, and for which there currently exists no other algorithm. Additionally, we have shown that our complete algorithm for searchlight scheduling can be directly extended to the $\phi$-Searchlight Scheduling Problem in which sensors have finite fields of view.

In the future, we hope that NP-hardness of the Searchlight Scheduling Problem can be shown, or else that the computational time complexity bounds for a complete algorithm can be improved. There are also many interesting and unexplored variations of the Searchlight Scheduling Problem. These include minimizing time to clear the environment, evaders with bounded speed, sensor limitations such as limited depth of field, and sensors sweeping a half-plane or cone through three dimensional environments.

# Chapter 4

# Distributed Searchlight Scheduling[1]

## 4.1 Introduction

Consider a group of robotic agents guarding a nonconvex polygonal environment, e.g., a floor plan. For simplicity, we model the agents as point masses. Each agent is equipped with a single unidirectional sweeping sensor called a *searchlight* (imagine a ray of light such as a laser range finder emanating from each agent). A searchlight aims only in one direction at a time and cannot penetrate the boundary of the environment, but its direction can be changed continuously by the agent. A point is detected by a searchlight at some instant if and only if the point lies on the ray. An evader is any point which can move continuously with unbounded speed. The *Searchlight Scheduling Problem* is to

> Find a schedule to rotate a set of stationary searchlights such that any evader in an environment will necessarily be detected in finite time.

[1] **© 2007 IEEE. Reprinted from [85] with permission of IEEE.**

A searchlight problem instance consists of an environment and a set of stationary guard positions. Obviously there can only exist a search schedule if all points in the environment are visible by some guard. For a graphical description of our objective, see Fig. 4.1 and 3.1.

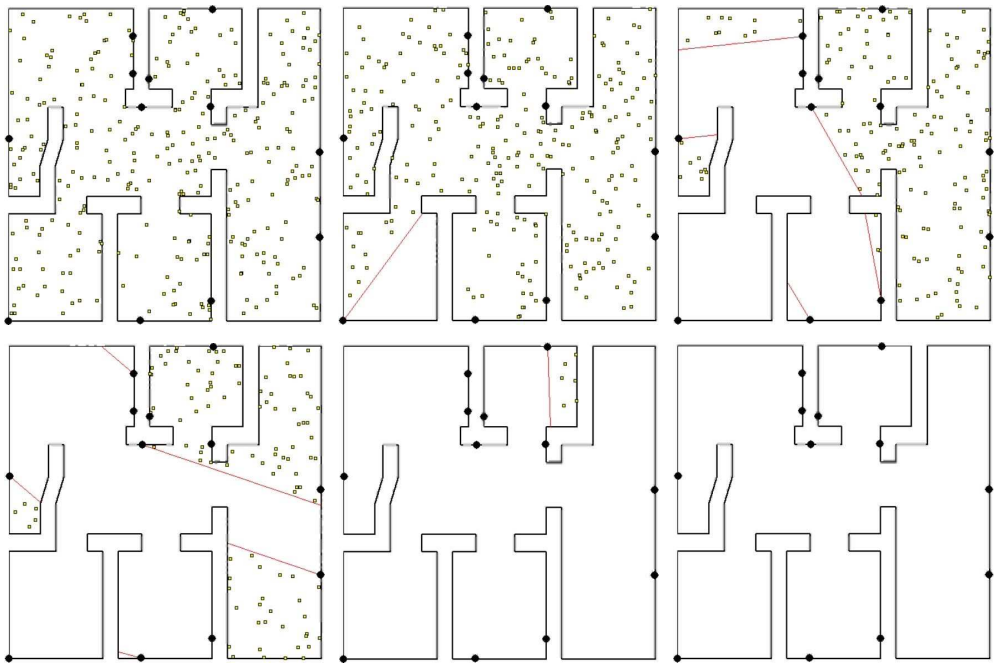


Figure 4.1. Simulation results of the PTSS algorithm described in Section 4.4.2, executed by agents (black dots) in a polygon shaped like a typical floor plan. Left to right, top to bottom, moving evaders (small yellow squares) disappear as they are detected by searchlights (red). The cleared region grows until it encompasses the entire environment.

To our knowledge the searchlight scheduling problem was first introduced in the inspiring work by Sugihara, Suzuki and Yamashita in [25], which considers simple polygonal environments and stationary searchlights. The work in [26] extends [25] by considering guards with multiple searchlights (they call a guard possessing $k$ searchlights a *$k$-searcher*) and polygonal environments containing holes. Some

works involving mobile searchlights, sometimes calling them *flashlights* or *beam detectors*, are [27], [28], [29], and [30]. Closely related is the Art Gallery Problem, namely the problem of finding a minimum set of locations from which the entire polygon is visible. Many variations on the Art Gallery Problem are wonderfully surveyed in [6], [7], and [8]. With an emphasis on practical imaging considerations, [16] describes a centralized task-specific procedure for choosing the locations of cameras in a network.

Assume now that each member of the group of guards is equipped with an omnidirectional line-of-sight sensor. By a line-of-sight sensor, we mean any device or combination of devices that can be used to determine, in its line-of-sight, (i) the position or state of another guard, and (ii) the distance to the boundary of the environment. By omnidirectional, we mean that the field-of-vision for the sensor is $2\pi$ radians. There exist distributed algorithms to deploy asynchronous mobile robots with such omnidirectional sensors into nonconvex environments, and they are guaranteed to converge to fixed positions from which the entire environment is visible, e.g., [22], [23], and the algorithm in Chapter 2. The algorithms in [23] and Chapter 2 guarantee the ancillary benefit of the final guard positions having a connected visibility graph ([23]). Once a set of guards seeing the entire environment has been established, it may be desired to continuously sweep the environment with searchlights so that any evader will be detected in finite time.

The main contribution of this chapter is the development of two asynchronous distributed algorithms to solve the searchlight scheduling problem. Correctness and completion time bounds for nonconvex polygonal environments are discussed. The first algorithm, called the DOWSS (Distributed One Way Sweep Strategy, Section 4.4.1), is a distributed version of a known algorithm described originally

in [25], but it can be very slow in clearing the entire environment because only one searchlight may rotate at a time. On-line processing time required by agents during execution of DOWSS is relatively low, so that the expedience with which an environment can be cleared is essentially limited by the maximum angular speed searchlights may be rotated at. In an effort to reduce the time to clear the environment, we develop a second algorithm, called the PTSS (Parallel Tree Sweep Strategy, Section 4.4.2), which sweeps searchlights in parallel if guards are placed in appropriate locations. These locations are related to an environment partition with certain properties. That we analyze the time it takes to clear an environment, given a bound on the angular rotation velocity, is a unique feature among all work involving searchlights to date.

We begin with some technical definitions, statement of assumptions, and brief description of the known centralized algorithm called the one way sweep strategy (appears, e.g., in [25], [26], [28]). We then develop a partially asynchronous model, a distributed one way sweep strategy, and our new algorithm the parallel tree sweep strategy.

## 4.2 Preliminaries

### 4.2.1 Notation

We begin by introducing some basic notation. We let $\mathbb{R}$, $\mathbb{S}^1$, and $\mathbb{N}$ represent the set of real numbers, the circle, and natural numbers, respectively. Given two points $x, y \in \mathbb{R}^2$, we let $[x, y]$ signify the *closed segment* between $x$ and $y$. Similarly, $]x, y[$ is the *open segment* between $x$ and $y$, $[x, y[$ represents the set $]x, y[\cup\{x\}$ and $]x, y]$ is the set $]x, y[\cup\{y\}$. Also, we shall use $P$ to refer to tuples

of elements in $\mathbb{R}^2$ of the form $(p^{[0]}, \ldots, p^{[N-1]})$ (these will be the locations of the agents), where $N$ denotes the total number of agents.

We now turn our attention to the environment we are interested in and to the concepts of visibility. Let $\mathcal{E}$ be a simple polygonal environment, possibly nonconvex. By simple, we mean that $\mathcal{E}$ does not contain any hole and the boundary does not intersect itself. Throughout this chapter, $n$ will refer to the number of edges of $\mathcal{E}$ and $r$ the number of reflex vertices. A point $a \in \mathcal{E}$ is *visible from* $b \in \mathcal{E}$ if $[a, b] \subset \mathcal{E}$. The *visibility set* $\mathcal{V}(p) \subset \mathcal{E}$ from a point $p \in \mathcal{E}$ is the set of points in $\mathcal{E}$ visible from $p$. A *visibility gap* of a point $p$ with respect to some region $R \subset \mathcal{E}$ is defined as any line segment $[a, b]$ such that $]a, b[ \subset \text{int}(R)$, $[a, b] \subset \partial\mathcal{V}(p)$, and it is maximal in the sense that $a, b \in \partial R$ (intuitively, visibility gaps block off portions of $R$ not visible from $p$). The visibility graph $\mathcal{G}_{\text{vis}}$ of a set of agents $P$ in environment $\mathcal{E}$ is the undirected graph with $P$ as the set of vertices and an edge between two agents if and only if they are visible to each other.

We now introduce some notation specific to the searchlight problem. An instance of the searchlight problem can be written as a pair $(\mathcal{E}, P)$, where $\mathcal{E}$ is an environment and $P$ is a set of agent locations. For convenience, we will refer to the searchlight of the $i$th agent as $s^{[i]}$ (which is located at $p^{[i]} \in \mathbb{R}^2$), and $S = \{s^{[0]}, \ldots, s^{[N-1]}\}$ will be the set of all searchlights. $\theta^{[i]}$ will also denote the angle of the $i$th searchlight in radians from the positive horizontal axis. So, if we say, e.g., aim $s^{[i]}$ at point $x$, what we really mean is set $\theta^{[i]}$ equal to an angle such that the $i$th searchlight is aimed at $x$. Searchlights do not block visibility of other searchlights.

The next few definitions were taken from [25].

**Definition 4.1** (schedule)**.** *The* schedule *of a searchlight $s^{[i]} \in S$ is a continuous*

*function* $\theta^{[i]} : [0, t^*] \mapsto \mathbb{S}^1$*, where* $[0, t^*]$ *is an interval of real time.*

The *ray* of $s^{[i]}$ at time $t \in [0, t^*]$ is the intersection of $\mathcal{V}(p^{[i]})$ and the semi-infinite ray starting at $p^{[i]}$ with direction $\theta^{[i]}(t)$. $s^{[i]}$ is said to be *aimed* at a point $x \in \mathcal{E}$ in some time instant if $x$ is on the ray of $s^{[i]}$. A point $x$ is *illuminated* if there exists a searchlight aimed at $x$.

**Definition 4.2** (separability)**.** *Two points in* $\mathcal{E}$ *are* separable *at time* $t \in [0, t^*]$ *if every curve connecting them in the interior of* $\mathcal{E}$ *contains an illuminated point, otherwise they are* nonseparable*.*

**Definition 4.3** (contamination and clarity)**.** *A point* $x \in \mathcal{E}$ *is* contaminated *if an undetected evader can be located at* $x$*, otherwise* $x$ *is* clear*. A region is said to be* contaminated *if it contains a contaminated point, otherwise it is* clear*.*

**Definition 4.4** (search schedule)**.** *Given* $\mathcal{E}$ *and a set of searchlight locations* $P = \{p^{[0]}, \ldots, p^{[N-1]}\}$*, the set* $\Theta = \{\theta^{[0]}, \ldots, \theta^{[N-1]}\}$ *is a* search schedule for $(\mathcal{E}, P)$ *if* $\mathcal{E}$ *is clear at* $t^*$*.*

### 4.2.2 Problem description and assumptions

We introduce the problem of interest. The *Distributed Searchlight Scheduling Problem* is to

> Design a distributed algorithm for a network of autonomous robotic agents in fixed positions, who will coordinate the rotation of their searchlights so that any evader in an environment will necessarily be detected in finite time. Furthermore, these agents are to operate using only information from local sensing and limited communication.

What is precisely meant by local sensing and limited communication will become clear in later sections. We make the following *main assumptions* about every searchlight instance in this chapter:

(i) The environment is a simple polygon with finitely many reflex vertices.

*Comments:* Compactness is a practical assumption for sensor range limitations. Simple connectedness means no holes. Having only finitely many reflex vertices precludes problems such as arise from fractal environments and will be important for proving the algorithms terminate in finite time.

(ii) Every point in the environment is visible from some agent and there are a finite number $N \in \mathbb{N}$ of agents.

*Comments:* If there were some point in the environment not visible by any agent, then an evader could remain there undetected for infinite time.

(iii) For every connected component of $\mathcal{G}_{\text{vis}}$, there is at least one agent located on the boundary of the environment.

*Comments:* This will be important for proving the algorithms terminate without failure. It also implies every agent is either on the boundary of the environment or visible from some other agent. If there existed an agent $i$ located at a point $p_i$ in the interior of the environment and not visible by any other agent, then there would exist $\epsilon > 0$ such that $\overline{B}_\epsilon(p_i) \cap \mathcal{V}(p_j) = \emptyset$ for $i \neq j$. An evader could thus evade detection by remaining in $\overline{B}_\epsilon(p_i)$ and simply staying on the opposite side of agent $i$ as $l_i$ points.

### 4.2.3 One Way Sweep Strategy (OWSS)

This section describes informally the centralized recursive One Way Sweep Strategy (OWSS hereinafter) originally introduced in [25]. The reader is referred to [25] for a detailed description. Centralized OWSS also appears in [28] and [26]. OWSS is a method for clearing a subregion of a simple 2D region $\mathcal{E}$ determined by

the rays of searchlights. The subregions of interest are the so-called *semiconvex subregions* of $\mathcal{E}$ *supported* by a set of searchlights at a given time and are defined as follows:

**Definition 4.5** (semiconvex subregion)**.** *$\mathcal{E}$ is always a semiconvex subregion of $\mathcal{E}$ supported by $\emptyset$. Furthermore, any $R \subset \mathcal{E}$ is a semiconvex subregion of $\mathcal{E}$ supported by a set of searchlights $S_{\text{sup}}$ if both of the following hold:*

*(i) It is enclosed by a segment of $\partial\mathcal{E}$ and the rays of some of the searchlights in $S$.*

*(ii) The interior of $R$ is not visible from any searchlight in $S$.*

The term "semiconvex" comes from the fact that any reflex vertex of a semiconvex subregion is also a reflex vertex of $\mathcal{E}$. In polygonal environments, all semiconvex subregions are polygons. The schedule used in Fig. 3.1 was based on OWSS, but as a more general example, consider Fig. 4.2. To clear the environment $\mathcal{E}$, which is a semiconvex subregion supported by $\emptyset$, we may begin by selecting an arbitrary searchlight on the boundary, say $s^{[0]}$. The first searchlight selected to clear an environment will be called the *root*. $s^{[0]}$ aims as far clockwise (cw hereinafter) as possible so that it is aligned along the cw-most edge. $s^{[0]}$ will then rotate couterclockwise (ccw hereinafter) through the environment, stopping incrementally whenever it encounters a visibility gap. The only visibility gap $s^{[0]}$ encounters produces the semiconvex subregion $R$ (thick border). At this time, another searchlight which sees across the visibility gap and is not in the interior of $R$, in this case $s^{[1]}$, is chosen to begin sweeping the area in $R$ not seen by $s^{[0]}$. Notice we have marked angles $\phi_{\text{start}}$ and $\phi_{\text{finish}}$. These are the cw-most and ccw-most directions, resp., in which $s^{[1]}$ can aim at some point in $R$. $s^{[1]}$ will rotate from $\phi_{\text{start}}$ to $\phi_{\text{finish}}$ and in the process encounter visibility gaps, each producing

the semiconvex subregions $R_1$, $R_j$, and $R_m$, which must be cleared by $s^{[2]}$ and/or $s^{[3]}$. As soon as $R$ is clear (when $s^{[1]} = \phi_{\text{finish}}$), $s^{[0]}$ can continue rotating until it is pointing along the wall immediately to its left at which time the entire environment is clear. The recursive nature of OWSS should be apparent at this point. Note that in OWSS (and DOWSS described later) it is actually arbitrary whether a searchlight rotates cw or ccw over a semiconvex subregion, but to simplify the discussion we always use ccw.

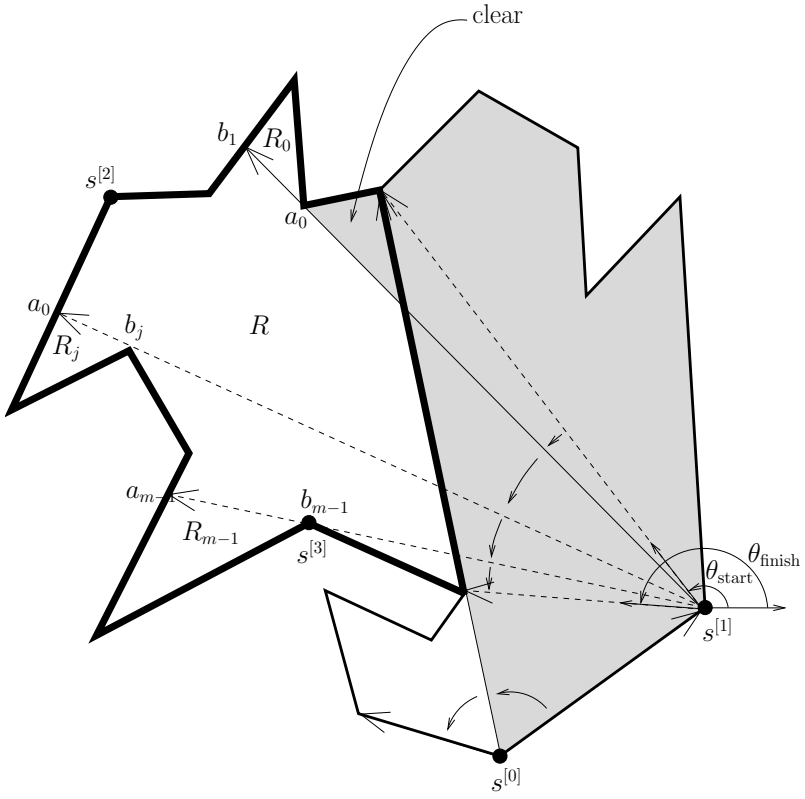



Figure 4.2. One Way Sweep Strategy (OWSS) clears, by rotating $s^{[1]}$, the semiconvex subregion $R$ (thick border) supported by $s^{[0]}$. $s^{[1]}$ must stop incrementally at each of its visibility gaps $[a_1, b_1]$, $[a_j, b_j]$, and $[a_m, b_m]$. In this recursive process, the regions ($R_1$, $R_j$, $R_m$) behind the visibility gaps become semiconvex subregions supported by $\{s^{[0]}, s^{[1]}\}$, and must be cleared using only the remaining searchlights ($s^{[2]}$ and $s^{[3]}$).

## 4.3 Asynchronous Network Agents

In this section we lay down the sensing and communication framework for agents with searchlights. Each agent is able to sense the relative position of any point in its visibility set as well as identify visibility gaps on the boundary of its visibility set. The agents' communication graph $\mathcal{G}_{\text{comm}}$ is assumed connected. An agent can rotate its searchlight continuously in any direction and turn it on or off.

Each of the $N$ agents has a unique identifier (UID), say $i$, and a portion of memory dedicated to outgoing messages with contents denoted by $\mathcal{M}^{[i]}$. Agent $i$ can broadcast its UID together with $\mathcal{M}^{[i]}$ to all agents within its communication region (defined differently in each algorithm). We assume a bounded time delay, $\delta > 0$, between a broadcast and the corresponding reception.

Each agent repeatedly performs the following sequence of actions between any two wake-up instants:

(i) SPEAK, that is, send a BROADCAST repeatedly at $\delta$ intervals, until it starts rotating;

(ii) LISTEN for a time interval at least $\delta$;

(iii) PROCESS and LISTEN after receiving a valid message;

(iv) ROTATE to an angle decided during PROCESS.

See Figure 4.3 for a schematic illustration of the schedule.

Any agent $i$ performing the ROTATE action does so according to the discrete-time control system

$$\theta^{[i]}(t + \Delta t) = \theta^{[i]}(t) + u^{[i]}, \tag{4.1}$$

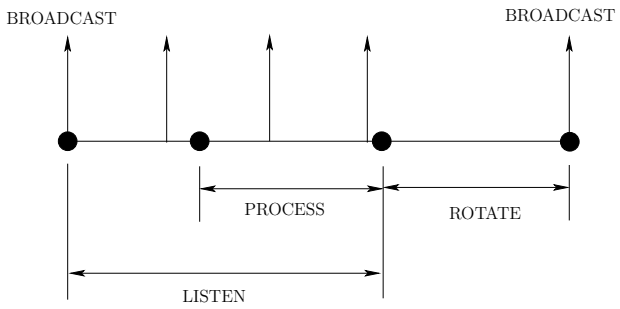



Figure 4.3. Sequence of actions performed by an agent $i$ in between two wake-up instants. Note that a BROADCAST is an instantaneous event taking place where there is a vertical pulse, where as the PROCESS, LISTEN and ROTATE actions take place over an interval. The ROTATE interval may be empty if the agent does not sweep.

where the control is bounded in magnitude by $u_{\max}$. The control action depends on time, values of variables stored in local memory, and the information obtained from communication and sensing. The subsequent wake-up instant is the time when the agent stops performing ROTATE and is not predetermined. This network model is identical to that used for distributed deployment in [22] and [23].

## 4.4 Distributed Algorithms

Here we design distributed algorithms for a network of agents as described above, where no agent has global knowledge of the environment or locations of all other agents.

### 4.4.1 Distributed One Way Sweep Strategy (DOWSS)

Once one understands OWSS as in Section 4.2.3, especially its recursive nature, performing one way sweep of an environment in a distributed fashion is fairly straightforward. We give here an informal description and supply a pseudocode in Table 4.1 (A more detailed pseudocode, which we refer to in the proofs, can be found in Appendix A). In our discussion root/parent/child will refer to the relative location of agents in the simulated one way sweep recursion tree. In this tree, each node corresponds to a one way rotation action by some agent. A single agent may correspond to more than one node, but only one node at a time. To begin DOWSS, some agent (the root[2]), say $i$, can aim as far clockwise as possible and then begin rotating until it encounters a visibility gap. Paused at a visibility gap, agent $i$ broadcasts a call for help to the network. For convenience, call the semiconvex subregion which $i$ needs help clearing $R$. All agents not busy in the set of supporting searchlight $S_{\text{sup}}$ (indeed at the zeroth level of recursion only the root is in $S_{rmsup}$), who also know they can see a portion of $\text{int}(R)$ but are not in $\text{int}(R)$, volunteer themselves to help $i$. Agent $i$ then chooses a child and the process continues recursively. In DOWSS as in Table 4.1, an agent needing help always chooses the first child to volunteer, but some other criteria could be used, e.g., who sees the largest portion of $R$. Whenever a child is finished helping, i.e., clearing a semiconvex subregion, it reports to its parent so the parent knows they may continue rotating.

The only subtle part of DOWSS is getting agents to recognize, without global knowledge of the environment, that they see the interior of a particular semicon-

[2] The root could be chosen by any leader election scheme, e.g., a predetermined or lowest UID.

vex subregion which some potential parent needs help clearing. More precisely, suppose some agent $l$ must decide whether to respond as a volunteer to agent $i$'s help request to clear a semiconvex subregion $R$. Agent $l$ must calculate if it actually satisfies $p^{[l]} \notin \text{int}(R)$ and $\text{int}(R) \cap \mathcal{V}(p^{[l]}) \neq \emptyset$. This is accomplished by agent $i$ sending along with its help request an oriented polyline $\psi$ (see SPEAK section of Table 4.1). By an oriented polyline we mean that $\psi$ consists of a set of points listed according to some orientation convention, e.g., so that if one were to walk along the points in the order listed, then the interior of $R$ would always be to the right. The polyline encodes the portion of $\partial R$ which is not part of $\partial\mathcal{E}$ and the orientation encodes which side of $\psi$ is the interior of $R$. Notice that for this to work, all agents must have a common reference frame. Whenever the root broadcasts a polyline, it is just a line segment, but as recursion becomes deeper, an agent needing help may have to calculate a polyline consisting of a portion of its own beam and its parent's polyline. The polyline may even close on itself and create a convex polygon. Examples of these scenarios are illustrated by in Fig 4.4. We conclude our description of DOWSS with the following theorem.

**Theorem 4.1** (Correctness of DOWSS)**.** *Given a simple polygonal environment $\mathcal{E}$ and agent positions $P = (p^{[0]}, \ldots, p^{[N-1]})$, let the following conditions hold:*

*(i) the standing assumptions are satisfied;*

*(ii) all agents $i \in \{0, \ldots, N-1\}$ have a common reference frame;*

*(iii) $p^{[0]} \in \partial\mathcal{E}$;*

*(iv) the agents operate under DOWSS.*

*Then $\mathcal{E}$ is cleared in finite time.*

*Proof.* As in Theorem 2 of [25], whenever an agent, say $i$, needs help clearing a semiconvex subregion $R$, there is some available agent $l$ satisfying $p^{[l]} \notin \text{int}(R)$

and $\text{int}(R) \cap \mathcal{V}(p^{[l]}) \neq \emptyset$. This comes from the standing assumption that for every connected component of $\mathcal{G}_{\text{vis}}$, there is at least one agent on the environment boundary. Now since visibility sets are closed, we may demand additionally that agent $l$ sees a portion of the oriented polyline $\psi$ sent to it by $i$. This means that in an execution of DOWSS, some $l$ will always be able to recognize, using only knowledge of $\mathcal{V}(p^{[l]})$ and $\psi$ from local sensing and limited communication, that it is able to help. We conclude DOWSS simulates OWSS. □

We now give an upper bound on the time it takes DOWSS to clear the environment assuming the searchlights rotate at some constant angular velocity $\omega$, and that communication and processing time are negligible.

**Lemma 4.1** (DOWSS Time to Clear Environment)**.** *Let agents in a network executing DOWSS rotate their searchlights with angular speed $\omega$. Then the time required to clear an environment with $r$ reflex vertices is no greater than $\frac{2\pi}{\omega}\frac{1-r^N}{1-r}$.*

*Proof.* There are only finitely many ($r$) reflex vertices of $\mathcal{E}$, and finitely many guards ($N$). Recall each visibility gap encountered during an execution of DOWSS produces a semiconvex subregion whose reflex vertices necessarily are part of $\partial\mathcal{E}$. This means the number of visibility gaps encountered by any agent when sweeping from $\phi_{\text{start}}$ to $\phi_{\text{finish}}$ (at any level of the recursion tree) can be no greater than $r$, i.e., refering to line 8 of PROCESS in DOWSS pseudocode Appendix A, $|G| = m \leq r$. Since the number of agents available to sweep a semiconvex subregion decreases by one for each level of recursion, the maximum depth of the recursion tree is upper bounded by $N-1$. It is apparent the number of nodes in the recursion tree cannot exceed $1 + r + r^2 + \cdots + r^{N-1} = \frac{1-r^N}{1-r}$. □

It is not known whether this bound is tight, but at leastq examples as in Fig. 4.5 can be constructed where DOWSS and OWSS run in $\mathcal{O}(r^2)$ ($\Rightarrow \mathcal{O}(n^2)$) time if guards are chosen malevolently. A key point is that DOWSS and OWSS do not specify (i) how to place guards given an environment, or (ii) how to optimally

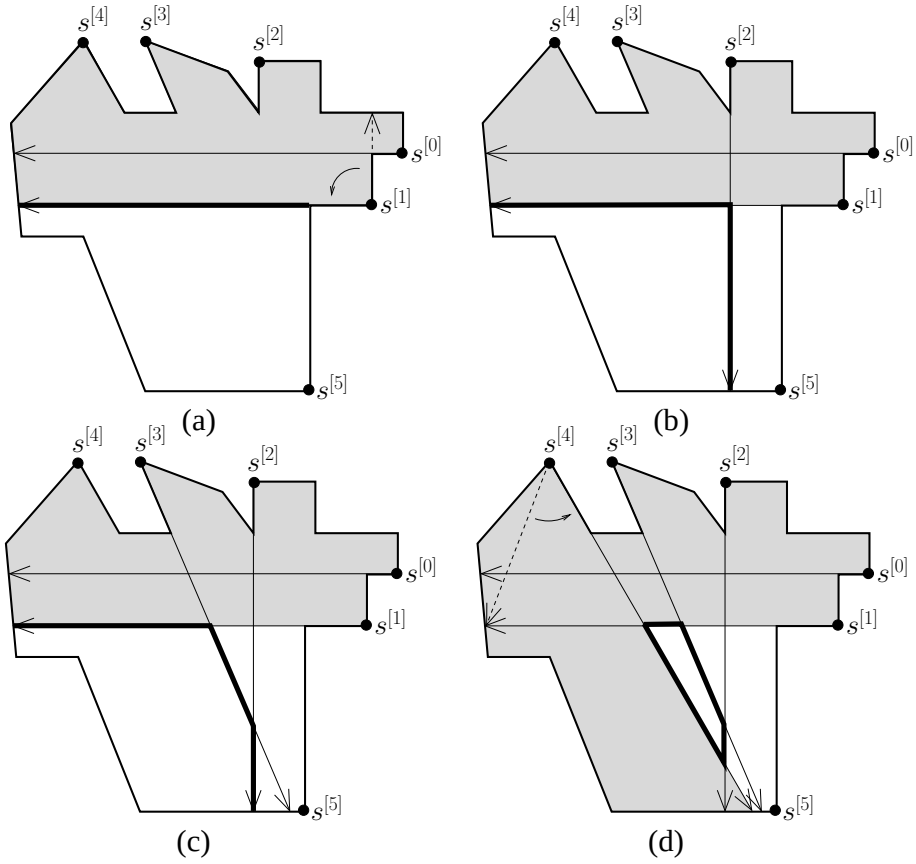



Figure 4.4. An example execution of DOWSS. The configuration in (a) results from $s^{[0]}$ clearing the very top of the region with help of $s^{[2]}$, $s^{[3]}$, and $s^{[4]}$ followed by $s^{[1]}$ attempting to clear the semiconvex subregion below where $s^{[0]}$ is aimed. When $s^{[1]}$ gets stuck, it requests help by broadcasting the thick black polyline in (a), in this case just a line segment. $s^{[2]}$ then helps $s^{[1]}$ but gets stuck right off, so it broadcasts the thick black polyline shown in (b). Next $s^{[3]}$ helps $s^{[2]}$ but gets stuck and broadcast the polyline in (c). Similarly $s^{[4]}$ broadcasts the polyline in (d), in this case a convex polygon, which only $s^{[5]}$ can clear. In general, information passed between agents during any execution of DOWSS will be in the form of either an oriented line segment (a), a general oriented polyline (b and c), or a convex polygon (d).

choose guards at each step given a set of guards. These are interesting unsolved problems in their own right which we do not explore in this chapter.

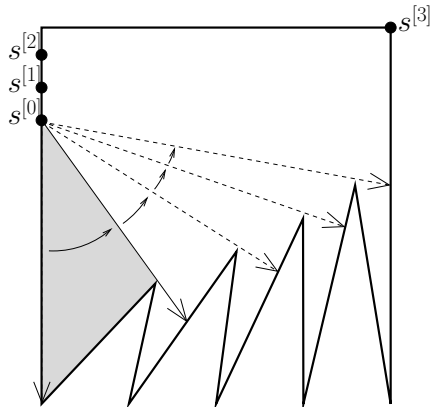



Figure 4.5. An example from a class of searchlight instances for which malevolent guard choice in OWSS or DOWSS implies time to clear the environment is $\mathcal{O}(r^2)$ (and therefore $\mathcal{O}(n^2)$). Here $r = 4$ reflex vertices are oriented on the bottom so that $s^{[r]} = s^{[4]}$ in the upper right corner sees the entire environment. $r - 1 = 3$ guards are placed in the upper left and $s^{[0]}$ is chosen as the root. $s^{[0]}$ clears up to the first reflex vertex (grey) where it stops and calls upon $s^{[1]}$ for help. $s^{[1]}$ then calls upon $s^{[2]}$ which likewise calls upon $s^{[3]}$. This happens every time $s^{[0]}$ stops (dashed lines) at the other $r - 1$ reflex vertices. The recursion tree of such an execution has $1 + r(r - 1)$ nodes, thus the environment is cleared in $\mathcal{O}(r^2)$ time.

Another performance measure of a distributed algorithm is the size of the messages which must be communicated.

**Lemma 4.2** (DOWSS Message Size)**.** *If the environment has $n$ sides, $r$ reflex vertices, and $N$ agents then the polyline (passed as a message between agents during DOWSS) consists of a list of no more than $\min\{r + 1, N\}$ points in $\mathbb{R}^2$. Furthermore, since $r \leq n - 3$, the list consists of no more than $n - 2$ points in $\mathbb{R}^2$.*

*Proof.* For every segment (which is a segment of some searchlight's beam) in such a polyline, there corresponds a unique reflex vertex of the environment. The correspondence comes from the fact that at a given time every searchlight supporting a semiconvex subregion has it's searchlight aimed at a reflex vertex where it's visibility is occluded. The uniqueness comes from the fact that if two

searchlights support the same semiconvex subregion, say $R$, and are aimed at the same reflex vertex, then only one of the searchlights' beams can actually constitute a portion of $\partial R$ of positive length. This shows the polyline can consist of no more than $r$ segments and therefore $r+1$ vertices. Also, in the worst case, the polyline grows by one edge for each level of recursion. Such polylines start out as a line segment (defined by two points) and the recursion depth cannot exceed $N-1$. We conclude the maximum number of points defining any polyline is $\min\{r+1, N\}$. □

That DOWSS allows flexibility in guard positions (only standing assumptions required) may be an advantage if agents are immobile. However, DOWSS only allowing one searchlight rotate at a time is a clear disadvantage when time to clear the environment is to be minimized. This lead us to design the algorithm in the next section.

### 4.4.2 Positioning Guards for Parallel Sweeping

The DOWSS algorithm in the previous section is a distributed message-passing and local sensing scheme to perform searchlight scheduling given *a priori* the location of the searchlights. Given an arbitrary positioning, time to completion of DOWSS can be large; see Lemma 4.1 and Figure 4.5. The algorithm we design in this section, called the Parallel Tree Sweep Strategy (PTSS), provides a way of choosing searchlight locations and a corresponding schedule to achieve faster clearing times. PTSS works roughly like this: According to some technical criteria described below, the environment is partitioned into regions called cells with one agent located in each cell. Additionally, the network possesses a distributed representation of a rooted tree. By distributed representation we mean that every agent knows who its parent and children are. Using the tree, agents rotate

Table 4.1. Asynchronous Schedule for DOWSS (cf Fig. 4.2, 4.3, 4.4)

| | |
|---|---|
| **Name:** | DOWSS |
| **Goal:** | Agents in the network coordinate their searchlight rotating to clear an environment $\mathcal{E}$. |
| **Assumes:** | Agents are stationary and have a completely connected communication topology with no packet loss. Sweeping is initialized by a root. |

For time $t > 0$, each agent executes the following actions between any two wake-up instants according to the schedule in Section 4.3:

**SPEAK**

Broadcast either

(i) a request for help,

(ii) a message to engage a child, or

(iii) a signal of task completion to a parent.

**LISTEN**

Listen for either

(i) a help request from a potential parent,

(ii) volunteers to help,

(iii) engagement by parent, or

(iv) current child reporting completion.

**PROCESS**

(i) Use oriented polyline from potential parent with information from sensing to check if able to help, or

(ii) if engaged, compute wayangles, visibility gaps and oriented polylines.

**ROTATE**

(i) Aim at start wayangle and switch searchlight on,

(ii) rotate to next wayangle, or

(iii) rotate to finish wayangle and switch searchlight off.

their searchlights in a way that expands the clear region from the root out to the leaves, thus clearing the entire environment. Since agents may operate in parallel, time to clear the environment is linear in the height of the tree and thus $\mathcal{O}(n)$. Guaranteed linear time to completion is a clear advantage over DOWSS which can be quadratic or worse (see Lemma 4.1 and Fig. 4.5). Before describing PTSS more precisely, we need a few definitions.

**Definition 4.6.** *(i) A set $\mathcal{S} \subset \mathbb{R}^2$ is* star-shaped *if there exists a point $p \in \mathcal{S}$ with the property that all points in $\mathcal{S}$ are visible from $p$. The set of all such points of a given star-shaped set $\mathcal{S}$ is called the* kernel *of $\mathcal{S}$ and is denoted by* $\ker(\mathcal{S})$.

*(ii) Given a compact subset $\mathcal{E}$ of $\mathbb{R}^2$, a* partition *of $\mathcal{E}$ is is a collection of sets $\{\mathcal{P}^{[0]}, \ldots, \mathcal{P}^{[N-1]}\}$ such that $\cup_{i=0}^{N-1} \mathcal{P}^{[i]} = \mathcal{E}$ where $\mathcal{P}^{[i]}$'s are compact, simply connected subsets of $\mathcal{E}$ with disjoint interiors. $\{\mathcal{P}^{[1]}, \ldots, \mathcal{P}^{[N]}\}$ will be called* cells *of the partition.*

For our purposes a *gap* (which visibility gap is a special case of) will refer to any segment $[q, q']$ with $q, q' \in \partial\mathcal{E}$ and $]q, q'[ \in \mathring{\mathcal{E}}$. The cells of the partitions we consider will be separated by gaps.

**Definition 4.7** (PTSS partition)**.** *Given a simple polygonal environment $\mathcal{E}$, a partition $\{\mathcal{P}^{[0]}, \ldots, \mathcal{P}^{[N-1]}\}$ is a* PTSS partition *if the following conditions are true:*

*(i) $\mathcal{P}^{[i]}$ is a star-shaped cell for all $i \in \{0, \ldots, N-1\}$;*

*(ii) the dual graph*[3] *of the partition is a tree;*

*(iii) a root, say $\mathcal{P}^{[0]}$, of the dual graph may be chosen so that $\ker(\mathcal{P}^{[0]}) \cap \partial\mathcal{E} \neq \emptyset$, and for any node other than the root, say $\mathcal{P}^{[k]}$ with parent $\mathcal{P}^{[j]}$, we have that $(\mathcal{P}^{[j]} \cap \mathcal{P}^{[k]}) \cap \ker(\mathcal{P}^{[k]}) \cap \partial\mathcal{E} \neq \emptyset$.*

[3] *The dual graph of a partition is the graph with cells corresponding to nodes, and there is an edge between nodes if the corresponding cells share a curve of nonzero length.*

**Definition 4.8.** *Given a PTSS partition* $\{\mathcal{P}^{[0]}, \ldots, \mathcal{P}^{[N-1]}\}$ *of* $\mathcal{E}$ *and a root cell* $\mathcal{P}^{[0]}$ *of the partition's dual graph satisfying the properties discussed in Definition 4.7, the corresponding (rooted)* PTSS tree *is defined as follows:*

*(i) the node set* $(p^{[0]}, \ldots, p^{[N-1]})$ *is such that* $p^{[0]} \in \ker(\mathcal{P}^{[0]}) \cap \partial\mathcal{E}$ *and for* $k > 1$, $p^{[k]} \in (\mathcal{P}^{[j]} \cap \mathcal{P}^{[k]}) \cap \ker(\mathcal{P}^{[k]}) \cap \partial\mathcal{E}$, *where* $\mathcal{P}^{[j]}$ *is the parent of* $\mathcal{P}^{[k]}$ *in the dual graph of the partition;*

*(ii) there exists an edge* $(p^{[j]}, p^{[k]})$ *if and only if there exists an edge* $(\mathcal{P}^{[j]}, \mathcal{P}^{[k]})$ *in the dual graph.*

We now describe two examples of PTSS partitions seen in Fig. 4.6. The left configuration in Fig. 4.6 results from what we call a Reflex Vertex Straddling (RVS hereinafter) deployment. RVS deployment begins with all agents located at the root followed by one agent moving to the furthest end of each of the root's visibility gaps, thus becoming children of the root. Likewise, further agents are deployed from each child to take positions on the furthest end of the children's visibility gaps located across the gaps dividing the parent from the children. In this way, the root's cell in the PTSS partition is just its visibility set, but the cells of all successive agents consist of the portion of the agents' visibility sets lying across the gaps dividing their cells from their respective parents' cells. It is easy to see that in final positions resulting from an RVS deployment, agents see the entire environment.

**Remark 4.1.** *Interestingly, in the RVS deployment, agents are deployed along a subset of the* critical angles *defined for the centralized searchlight scheduling algorithm of Chapter 3.*

**Lemma 4.3.** *RVS deployment requires, in general, no more than* $r + 1 \leq n - 2$ *agents to see the entire environment from their final positions. In an orthogonal environment, no more than* $\frac{n}{2} - 2$ *agents are required.*

*Proof.* Follows from the fact that in addition to the root, no more than one agent will be placed for each reflex vertex (only reflex vertices occlude visibility). □

See Fig. 4.1 for simulation results of PTSS executed by agents in an RVS configuration. The right configuration in Fig. 4.6 results from the deployment described in [23] in which an orthogonal environment is partitioned into convex quadrilaterals.

**Lemma 4.4.** *The deployment described in [23] requires no more than* $\frac{n}{2}-2$ *agents to see the entire (orthogonal) environment from their final positions.*

*Proof.* See [23]. □

Both of the PTSS configurations in these examples may be generated via distributed deployment algorithms in which agents perform a depth-first, breadth-first, or randomized search on the PTSS tree constructed on-line. Please refer to [22] and [23] for a detailed description of these algorithms.

We now turn our attention to the pseudocode in Table 4.2 (A more detailed pseudocode, which we refer to in the proofs, can be found in Appendix A) and describe PTSS more precisely. Suppose some agents are positioned in an environment according to a PTSS partition and tree with agent 1 as the root. PTSS begins by agent 1 pointing its searchlight along a wall in the direction $\phi_{\text{start}}$ and then rotating away from the wall toward $\phi_{\text{finish}}$, pausing whenever it encounters the first side of a gap, say $\phi_j$, where $j$ is odd. Paused at $\phi_j$, agent 1 sends a message to its child at that gap, say agent 2, so that agent 2 knows it should aim its searchlight across the gap. Once agent 2 has its searchlight safely aimed across the gap, it sends a message to agent 1 so that agent 1 knows it may continue rotating over the whole gap. When agent 1 has reached the other side of the gap

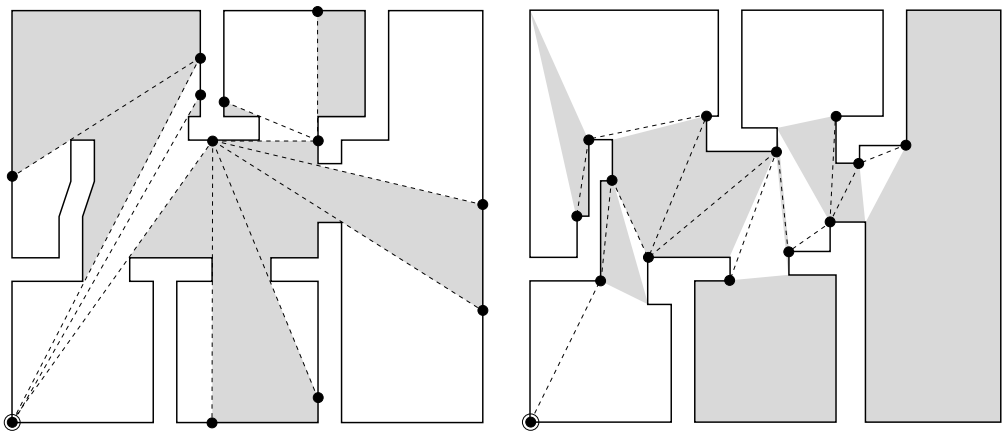


Figure 4.6. Left are agent positions resulting from a Reflex Vertex Straddling (RVS) deployment. Right are agent positions resulting from the deployment described in [23] in which an orthogonal environment is partitioned into convex quadrilaterals. The PTSS partitions are shown by coloring the cells alternating grey and white (caution: grey does not depict clarity here). Dotted lines show edges of the PTSS tree where the circled agent is the root.

at $\phi_{j+1}$, agent 1 sends a message to agent 2 and both agents continue clearing the rest of their cells concurrently, stopping at gaps and coordinating with children as necessary. In this way, the clear region expands from the root to the leaves at which time the entire environment has been cleared. We arrive at the following lemmas and correctness result.

**Lemma 4.5** (Expanding a Clear Region Across a Gap)**.** *Suppose an environment is endowed with a PTSS partition and tree, and that agent $i$ is a parent of agent $j$ (see Fig. 4.7). Then a clear region may always be expanded across the gap from $\mathcal{P}^{[i]}$ to $\mathcal{P}^{[j]}$ by $s^{[j]}$ first aiming across the gap and waiting for $s^{[i]}$ to rotate over the gap. Both agents may then continue clearing the remainder of their respective cells concurrently.*

*Proof.* This obviously hold for the scenario in Fig. 4.7. Using the definition of PTSS partition, it is clear any general PTSS parent-child relationship is reducible

to the case in Fig. 4.7. □

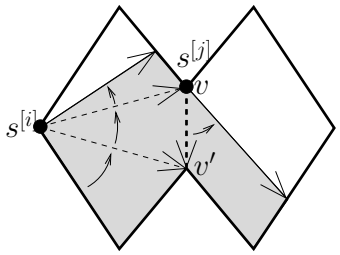



Figure 4.7. Expanding a clear region (grey) across a gap (thick dashed segment [v, v']) from cell $\mathcal{P}^{[i]}$ to cell $\mathcal{P}^{[j]}$ may always be accomplished by the child ($s^{[j]}$) aiming across the gap and waiting for the parent ($s^{[i]}$) to rotate over the gap. Both agents may then continue clearing the remainder of their respective cells.

**Theorem 4.2** (Correctness of PTSS)**.** *Given a simple polygonal environment $\mathcal{E}$ and agent positions*
$P = (p^{[0]}, \ldots, p^{[N-1]})$*, let the following conditions hold:*

*(i) the standing assumptions are satisfied;*

*(ii) all agents $i \in \{0, \ldots, N-1\}$ are positioned in a PTSS partition and rooted tree with agent 0 as the root;*

*(iii) the agents operate under PTSS.*

*Then $\mathcal{E}$ is cleared in finite time.*

*Proof.* Follows immediately from Lemma 4.5. □

Since multiple branches of the PTSS tree may be cleared concurrently, and using Lemmas 4.3 and 4.4, we have the next lemma (assuming processing and communication time are negligible, cf. Lemma 4.1).

**Lemma 4.6** (PTSS Time to Clear Environment)**.** *Let the agents in a network executing PTSS rotate their searchlights with angular speed $\omega$. Then time required to clear an environment is*

*(i) linear in the height of the PTSS tree;*

*(ii) no greater than $\frac{2\pi}{\omega}(r+1) \leq \frac{2\pi}{\omega}(n-2)$ if agents are in final positions according to an RVS deployment;*

*(iii) no greater than $\frac{\pi}{\omega}(n-2)$ if agents are in final positions in an orthogonal polygon according to an RVS deployment or the deployment described in [23].*

*Proof.* With communication time neglibile, each child will wait for it's parent a maximum time of $\frac{2\pi}{\omega}$. It now suffices to observe that the maximum length of any parent-child sequence is just the height of the PTSS tree. □

Looking at the SPEAK section of the PTSS pseudocode in Appendix A, it is easy to see that message size is constant (cf. Lemma 4.2).

**Lemma 4.7** (PTSS Message Size)**.** *Messages passed between agents executing PTSS have constant size.*

Requiring guards to be situated in a PTSS tree is more restrictive than the mere standing assumptions required by DOWSS, but the time savings using PTSS over DOWSS can be considerable. Despite our two example schemes to construct a PTSS tree, it is not clear how to construct one which clears an environment in minimum time among all possible PTSS trees. It is also not clear how to optimally choose the root of the tree (point of deployment). However, if the environment layout is known a priori and one may choose the root location, then an exhaustive strategy may be adopted whereby all possible root choices are compared.

Table 4.2. Asynchronous Schedule for PTSS (cf Fig. 4.3, 4.7, 4.6)

| | |
|---|---|
| **Name:** | PTSS |
| **Goal:** | Agents in the network coordinate their searchlight rotating to clear an environment $\mathcal{E}$. |
| **Assumes:** | Agents are statically positioned as nodes in a PTSS partition and tree, and each knows a priori the gaps of its cell and UIDs of the corresponding children and parent. Sweeping is initialized by the root. |

For time $t > 0$, each agent executes the following actions between any two wake-up instants according to the schedule in Section 4.3:

**SPEAK**

Broadcast either

(i) a command for a child to aim across a gap,

(ii) a confirmation to a parent when aimed across gap, or

(iii) when finished rotating over a gap, a signal of completion to the child.

**LISTEN**

Listen for either

(i) instruction from a parent to aim across a gap,

(ii) confirmation from a child aimed across a gap, or

(iii) confirmation that parent has passed the gap.

**PROCESS**

When first engaged, compute wayangles where coordination with children will be necessary.

**ROTATE**

(i) Aim at start wayangle and switch searchlight on,

(ii) rotate to next wayangle, or

(iii) rotate to finish wayangle and switch searchlight off.

## 4.5 Conclusion

In this chapter we have provided two solutions to the distributed searchlight scheduling problem. DOWSS requires that the guards satisfy the standing assumptions, has message size $\mathcal{O}(n)$, and sometimes requires time $\mathcal{O}(r^2)$ to clear an environment. PTSS requires that the agents be positioned according to a PTSS tree, has constant message size, and requires time linear in the height of the PTSS tree. We have given two procedures for constructing PTSS trees, one requiring no more than $r \leq n-3$ guards for a general polygonal environment, and two requiring no more than $\frac{n-2}{2}$ guards for an orthogonal environment. Guards rotate through a total angle no greater than $2\pi$, so the upper bounds on the time for PTSS to clear an environment with these partitions are $\frac{2\pi}{\omega}r \leq \frac{2\pi}{\omega}(n-3)$ and $\frac{\pi}{\omega}(n-2)$, respectively. Because PTSS allows searchlights to rotate concurrently, it generally clears an environment much faster than DOWSS. However, a direct comparison is not appropriate since DOWSS does not specify how to choose guards whereas PTSS does.

To extend DOWSS and PTSS for environments with holes, one simple solution is to add one guard per hole, where a simply connected environment is simulated by the extra guards using their beams to connect the holes to the outer boundary. Another straightforward extension for PTSS would be to combine it directly with a distributed deployment algorithm such as those in [22] and [23], so that deployment and searchlight rotation happen concurrently. This suggests an interesting problem we hope to explore in the future, namely minimizing the time to perform a coordinated search given a limited number of mobile guards. Other considerations for the future include loosening the requirements in the definition of the PTSS partition, and incorporating other sensor constraints such as limited

depth of field and beam incidence.

# Chapter 5

# Path Planning for a Visual Reconnaissance UAV[1]

## 5.1 Introduction

In this chapter we present novel path planning algorithms for a single fixed-wing aircraft performing a reconnaissance mission using EO (Electro-Optical) camera(s). Given a set of stationary ground targets in a terrain (natural, urban, or mixed), the objective is to compute a path for the reconnaissance aircraft so that it can photograph all targets in minimum time. That the targets are situated in terrain plays a significant role because terrain features can occlude visibility. As a result, in order for a target to be photographed, the aircraft must be located where both (1) the target is in close enough range to satisfy the photograph's resolution requirements, and (2) the line-of-sight between the aircraft and the target is not blocked by terrain. For a given target, we call the set of all such aircraft positions the target's *visibility region*. An example visibility re-

[1]**Reprinted from [93, 94, 95] with permission of the American Institute of Aeronautics and Astronautics.**

gion is illustrated in Fig. 5.1. In full generality, the aircraft path planning can be

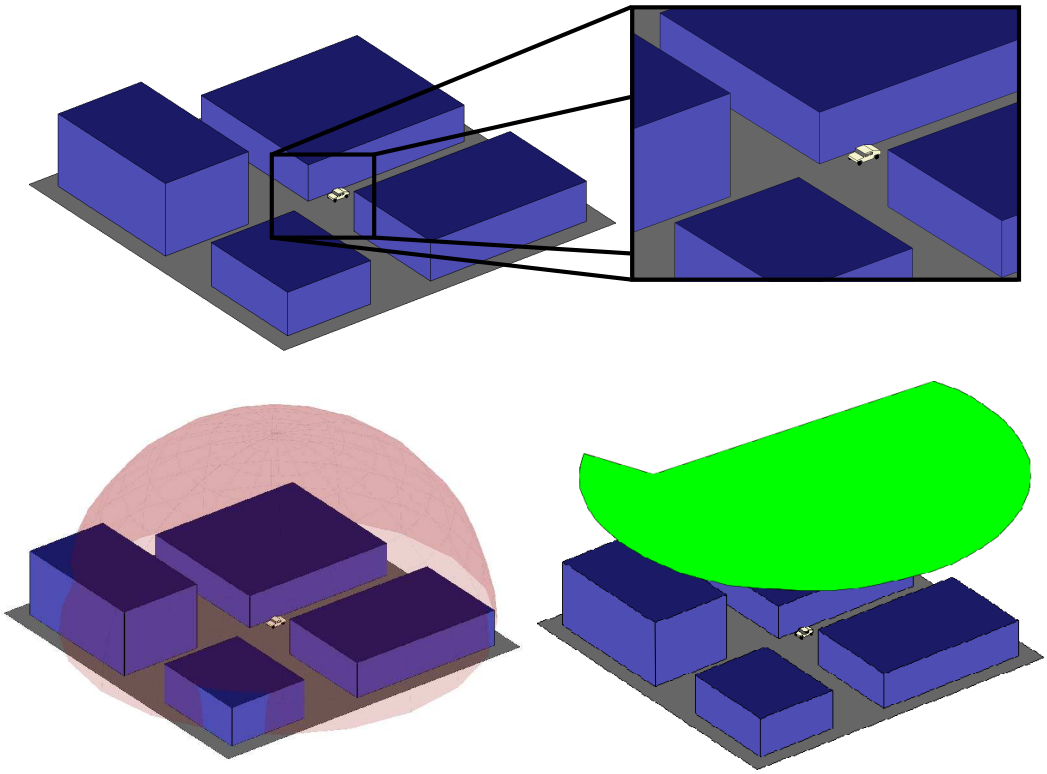


Figure 5.1. Top is shown an example target, a ground vehicle parked next to a building in urban terrain. The set of all points which are close enough to the target to satisfy photograph resolution requirements is a solid sphere (bottom left). The green two-dimensional region in the sky (bottom right) shows the subset of the sphere, at a reconnaissance aircraft's altitude $h$, where target visibility is not occluded by terrain. Assuming the aircraft body itself doesn't occlude visibility, then flying the aircraft through the green region is sufficient for the target to be photographed, hence we call it the target's *visibility region* for fixed aircraft altitude $h$.

complicated by wind, airspace constraints (e.g. due to enemy threats or collision avoidance), aircraft dynamic constraints, and the aircraft body itself occluding visibility. However, under simplifying assumptions, if we model the aircraft as a

Dubins vehicle[2], approximate the targets' visibility regions by polygons, and let the path be a closed tour (loop), then the reconnaissance path planning problem can be reduced to the following.

> For a Dubins vehicle, find a shortest planar closed tour which visits at least one point in each of a set of polygons.

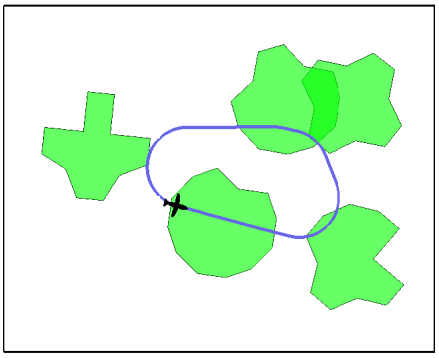


Figure 5.2. Example problem instance and candidate solution path for the *PVDTSP* (*Polygon-Visiting Dubins Traveling Salesman Problem*). In order to photograph all targets, the aircraft must fly through at least one point in each target's visibility region (green), cf. Fig. 5.1.

We refer to this henceforth as the *PVDTSP Polygon-Visiting Dubins Traveling Salesman Problem*) since it is a variation of the famous *TSP (Traveling Salesman Problem)*.[3] A graphical illustration of the PVDTSP is shown in Fig. 5.1.

To our knowledge the PVDTSP has not previously been studied. Because the PVDTSP has embedded in it the combinatorial problem of choosing the order

[2]A Dubins vehicle is one which moves only forward and has a minimum turning radius [45, 46].

[3]The TSP, one of the most famous NP-hard problems of combinatorial optimization, is to find a minimum cost tour (cyclic path) through a weighted graph such that every vertex is visited exactly once. If the graph is directed, it is called the ATSP (Asymmetric TSP). See, e.g., [96] for a brief introduction, or [59] for an extensive treatment.

to visit the polygons, the solution space is very large and discontinuous. This precludes direct application of numerical optimal control techniques traditionally used in trajectory optimization, surveyed, e.g., in [47]. However, several related variations of the TSP are of interest. The *ETSP (Euclidean TSP)* is a TSP where the vertices of the graph are points in the Euclidean plane $\mathbb{R}^2$ and the edges are weighted with Euclidean distances. In the *ETSPN (Euclidean TSP with Neighborhoods)* one seeks a shortest closed Euclidean path passing through $n$ subsets of the plane. The ETSP is NP-hard [48] and so is the ETSPN by virtue of being a generalization of the ETSP. The *DTSP (Dubins TSP)*, where a Dubins vehicle must follow a shortest tour through $n$ single point targets in the plane, is known to be NP-hard in $n$ [49]. Various heuristics for both single and multi-vehicle versions of the DTSP can be found, e.g., in [50], [51], and [52]. The PVDTSP reduces to the ETSPN in the limit as the vehicle's minimum turning radius becomes small compared to the distances between polygons. Similarly, as the area of the polygons goes to zero, the PVDTSP reduces to the DTSP, hence the PVDTSP is NP-hard. There exist a number of algorithms with approximation guarantees for both the DTSP [53, 54, 55] and ETSPN [56, 57, 58], but it appears that extending any of these algorithms to the PVDTSP would put undesirable restrictions on the problem instances which could be handled, e.g., the polygons would not be allowed to overlap. The *FOTSP (Finite One-in-set TSP)*[4] is the problem of finding a closed path of minimum cost which passes through at least one vertex in each of a finite collection of *clusters*, the clusters being mutually exclusive finite vertex sets. The FOTSP is NP-hard because it has as a special case the *ATSP (Asymmetric TSP)* [59]. An FOTSP instance can be solved exactly by transforming it into an ATSP

[4]What we have chosen to call the FOTSP is known variously in the literature as "Group-TSP", "Generalized-TSP", "One-of-a-Set TSP", "Errand Scheduling Problem", "Multiple Choice TSP", "Covering Salesman Problem", or "International TSP".

instance using the *Noon-Bean transformation* from [60], then invoking an ATSP solver. Alternatively, an FOTSP can be solved using an approximate dynamic programming technique as in [61]. In the robotics literature [18, 62], a *sampling-based roadmap method*[5] refers to any algorithm which operates by sampling a finite set of points from a continuous state space in order to reduce a continuous motion planning problem to planning on a finite discrete graph. Sampling-based roadmap methods have traditionally only been used for collision-free point-to-point path planning amongst obstacles, however, in [63] approximate solutions to the DTSP are found by sampling discrete sets of orientations that the Dubins vehicle can have over each target, essentially approximating a DTSP instance by an FOTSP instance. They then use the Noon-Bean transformation to convert the FOTSP instance into an ATSP instance so that a standard ATSP solver can be applied. Discretization of the vehicle state space in order to approximate the original problem by an FOTSP is a key idea which we build upon in designing sampling-based roadmap methods for the PVDTSP in the present work. For NP-hard problems such as the TSP and most of its variations, another possible approach is to use metaheuristic algorithms, e.g., tabu search, simulated annealing, or genetic algorithms [64]. These techniques typically lack performance guarantees, yet obtain good solutions in reasonable computation time. Particularly genetic algorithms have recently been applied to variations of the TSP and UAV motion planning problems [65, 66, 67, 68, 69, 70, 71]. *Genetic algorithm* is an umbrella term referring to any iterative procedure which mimics biological evolution by operating on a population of candidate solutions encoded as so-called chromosomes. The genetic operators of crossover and mutation are successively applied, generation

[5]In this usage, "method" means a high level algorithm having multiple components, each of which may be considered an algorithm in its own right.

after generation, until a sufficiently fit solution appears in the population. It is not obvious how to adapt existing genetic algorithms to the PVDTSP, nor is it clear whether any such adaptations would be effective.

There are four main contributions in this chapter. First, we precisely formulate the general aircraft visual reconnaissance problem for static ground targets in terrain. Under simplifying assumptions, we reduce our general formulation to the PVDTSP. Although the PVDTSP reduces to the well-studied DTSP and ETSP in the *sparse limit* as targets are very far apart, we provide a worst-case analysis demonstrating the importance of developing specialized algorithms for the PVDTSP in the *dense limit* as targets are close together and polygons may overlap significantly. Our second contribution is the design and numerical study of two sampling-based roadmap methods for the PVDTSP. These methods operate by sampling finite discrete sets of vehicle states to approximate a PVDTSP instance by an FOTSP instance, then applying existing FOTSP solving techniques. One of our sampling-based roadmap methods uses the Noon-Bean transformation from [60] and is *resolution complete*, which means it provably converges to a nonisolated global optimum as the number of samples grows. Our other sampling-based roadmap method achieves faster computation times by using the approximate dynamic programming technique from [61], but consequently only converges to a nonisolated global optimum modulo target order. While we have borrowed the idea of approximation by an FOTSP from [63], the present work goes beyond a simple extension in that we (1) illustrate the connection with sampling-based roadmap methods used for path planning in the robotics literature[6], (2) use a

[6]Although [63] appears to be the first application of a sampling-based roadmap method to a TSP-type problem, they do not use the term "sampling-based roadmap method", nor is there any mention of the connection with sampling-based roadmap methods in the robotics literature.

novel sampling technique to reduce computational time complexity, and (3) provide proof of convergence to nonisolated global optima. As a third contribution, we design a genetic algorithm for the PVDTSP. The genetic algorithm has no performance guarantees but is easiest to implement and tends to find good feasible solutions quickly. Numerical experiments indicate that both the sampling-based roadmap methods and genetic algorithm deliver good solutions suitably quickly for online purposes when applied to PVDTSP instances having up to about 20 targets. Additionally, all the algorithms have a means for a user to trade off computation time for solution quality. Our fourth contribution is to describe how the modular nature of all the algorithms allows them to easily be extended to handle wind, airspace constraints, any vehicle dynamics, and open-path (vs. closed-tour) problems.

This chapter is organized as follows. In Sec. 5.2 we introduce notation, mathematically formulate the minimum time reconnaissance aircraft path planning problem, show how to reduce the problem to a PVDTSP, and provide the worst-case analysis motivating the development of specialized PVDTSP algorithms. In Sec. 5.3 we present, analyze, and numerically validate the sampling-based roadmap methods. In Sec. 5.4 we present the genetic algorithm with a supporting Monte-Carlo numerical study. Finally, we describe how all our algorithms can be extended in Sec. 5.5 and conclude in Section 5.6.

## 5.2 Mathematical Formulation

We begin with some preliminary notation. The $s$-dimensional Euclidean space is $\mathbb{R}^s$ and $\mathbb{S}$ is the circle parameterized by angle radians ranging from 0 to $2\pi$, 0

and $2\pi$ identified. Let $\mathcal{T} = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n\}$ be the set of $n$ targets which must be photographed by our aircraft. Given a set $A$, we denote its cardinality by $|A|$, its interior by $\operatorname{int} A$, and its power set, i.e., the set of all subsets of $A$, by $2^A$. Given two sets $A$ and $B$, $A \times B$ is the Cartesian product of these sets. The complete state of our reconnaissance aircraft is encoded in a vector $\mathbf{x}$, which takes a value in the aircraft's state space $X$. We can segregate $\mathbf{x}$ into internal and external states so that

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_{\text{internal}} \\ \\ \mathbf{x}_{\text{external}} \end{bmatrix} \in X = X_{\text{internal}} \times X_{\text{external}}. \tag{5.1}$$

The internal state $\mathbf{x}_{\text{internal}}$ accounts for control surface states, and more importantly, if the aircraft has gimbaled camera(s), then also for the camera state(s). The external state $\mathbf{x}_{\text{external}}$ accounts for the aircraft body position and velocity in the full six degrees of freedom.

We now define a map $\mathcal{V} : \mathcal{T} \to 2^X$ from the set of targets to subsets of the aircraft state space. Under this map, $\mathcal{V}(\mathcal{T}_i) \subset X$, called the $i$th target's *visibility region*, is precisely the set of all aircraft states such that $\mathcal{T}_i$ can be photographed whenever the aircraft is in that state. Later, in Sec. 5.2.1, we discuss how to calculate visibility regions from a terrain model, but let us assume for now we can make this calculation. We also assume a BVP (Boundary Value Problem) solver is available which calculates the minimum time aircraft trajectory between any two states $\mathbf{x}$ and $\mathbf{x}'$, provided a trajectory exists. We treat this minimum time between states as a "black box" distance function denoted by $d(\mathbf{x}, \mathbf{x}')$. Now our

**minimum time reconnaissance path planning problem** can be stated as

$$\begin{aligned} \text{Minimize :} \quad & C(\mathbf{x}_1, \ldots, \mathbf{x}_n) \;=\; \textstyle\sum_{i=1}^{n-1} d(\mathbf{x}_i, \mathbf{x}_{i+1}) + d(\mathbf{x}_n, \mathbf{x}_1) \\ \text{Subject To :} \quad & \text{for each } i \in \{1, \ldots, n\} \text{ there exists } j \in \{1, \ldots, n\} \\ & \text{such that } \mathbf{x}_j \in \mathcal{V}(\mathcal{T}_i), \end{aligned} \tag{5.2}$$

where the decision variables are the states $\mathbf{x}_i$ $(i = 1, \ldots, n)$. Once an optimal sequence of states $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ has been chosen, then the minimum time state-to-state trajectory planner can be used to connect each pair of consecutive states, thus we obtain a minimum time closed reconnaissance tour. Since the complete state space of an aircraft can be very complicated, we simplify the discussion by making the following **main assumptions**.

(i) The aircraft is modeled as a Dubins vehicle with minimum turning radius $r_{\min}$, fixed altitude $h$, and constant airspeed $V_{\mathrm{a}}$.

*Comments:* Common for small low-power UAVs.

(ii) Regardless of state, the aircraft body never occludes visibility between the camera and a target.

*Comments:* Holds when either there are multiple cameras covering all angles from the aircraft, or there is a sufficiently flexible gimbaled camera with dynamics faster than the aircraft body dynamics.[7]

(iii) There are no airspace constraints nor wind.

*Comments:* As to be discussed in Sec. 5.5, our results can easily be extended to handle wind and no-fly zones.

[7] An omnidirectional camera is another possibility, but they typically have poor resolution.

In accordance with assumption (i), the aircraft dynamics take the form

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} V_{\mathrm{a}} \sin(\psi) \\ V_{\mathrm{a}} \cos(\psi) \\ u \end{bmatrix}, \tag{5.3}$$

where $(x, y) \in \mathbb{R}^2$ are earth-fixed Cartesian coordinates, $\psi \in \mathbb{S}$ is the azimuth angle, and $u$ is the input to an autopilot system. Assumption (ii) tells us that a target can be photographed independent of aircraft azimuth $\psi$, therefore we can abstract out $\mathbf{x}_{\text{internal}}$ so that the aircraft state space is reduced to

$$\mathbf{x} = (x, y, \psi) \ \in \ X = \mathbb{R}^2 \times \mathbb{S} = \mathrm{SE}(2), \tag{5.4}$$

and the Visibility sets $\mathcal{V}(\mathcal{T}_1), \ldots, \mathcal{V}(\mathcal{T}_n)$ are reduced to 2-dimensional regions in $\mathbb{R}^2$ as shown in Fig. 5.1 and 5.1 (as opposed to subsets of $X = \mathbb{R}^2 \times \mathbb{S}$). Hereinafter we refer to the state of a Dubins vehicle interchangeably as "state" or "pose" (position with orientation). The minimum time path between two Dubins states $\mathbf{x}$ and $\mathbf{x}'$ can be computed very quickly in constant time [45, 97]. This provides us with our "black box" distance function $d(\mathbf{x}, \mathbf{x}')$ as it appears in the optimization problem Eq. 5.2. Although visibility regions may contain circular arcs due to the camera range constraint, they can be well approximated by polygons. We have now reduced our minimum time reconnaissance path planning problem to a PVDTSP.

In some UAV systems in the field today, target visibility sets are neglected and reconnaissance paths are planned by simply solving the DTSP over the target positions, i.e., the UAV is restricted to pass directly over each target in order to photograph it. However the worst-case analysis in the following Theorem 5.1 demonstrates that an arbitrarily large relative cost increase can be incurred by

solving the DTSP instead of the PVDTSP. This cost increase is most pronounced in the *dense limit* (left in Fig. 5.2) as targets become very close together, which motivates our development of specialized PVDTSP algorithms for tight urban scenarios especially. In contrast, in the *sparse limit* (right in Fig. 5.2) when the minimum turning radius and visibility set diameters are much smaller than the distances between targets, there is no significant advantage to solving the PVDTSP over the DTSP nor over the ETSP.

**Theorem 5.1** (DTSP vs. PVDTSP Worst-Case Analysis)**.** *In a fixed compact subset of the plane* $\mathbb{R}^2$*, solving the DTSP over point targets instead of the PVDTSP over those same targets' visibility sets may incur a cost penalty of order* $\Omega(n)$ *in the worst case.*[8]

*Proof.* The set of all DTSP tours through $n$ point targets is a subset of all PVDTSP tours through those same targets' visibility sets, therefore the length of a tour that results from solving the PVDTSP to optimality can be no greater than that of solving the DTSP. Now it suffices to prove the theorem by demonstrating a class of visual reconnaissance problem instances, parameterized by the number of targets $n$, for which the tour cost when solved as a DTSP is order $\Omega(n)$ (lower bounded) yet only order $O(1)$ (upper bounded) when solved as a PVDTSP. One such class of instances is illustrated left in Fig. 5.2.[9] Given any $n$ noncolinear point targets in the plane, we can linearly scale them until the radius of the circle constructed from any three of them has radius smaller than the Dubins vehicle minimum turn radius. This scaling ensures that, in order to fly a feasible DTSP tour, the aircraft must travel a distance at least the length of one minimum turn radius circle for every two targets. Solving the DTSP over these points would thus cost $\Omega(n)$, yet letting the intersection of the targets visibility sets' contain all the targets, the PVDTSP could be solved with a single minimum turn radius

[8] *A function* $f(n)$ *is said to be* $\Omega(n)$ *if there exist positive constants* $c$ *and* $n_0$ *such that* $f(n) \geq cn$ *for all* $n \geq n_0$.

[9] Such a class of instances has been used previously in [54] to show DTSP tours in general have worst-case length $\Omega(n)$.

loop and thus cost only $O(1)$. □

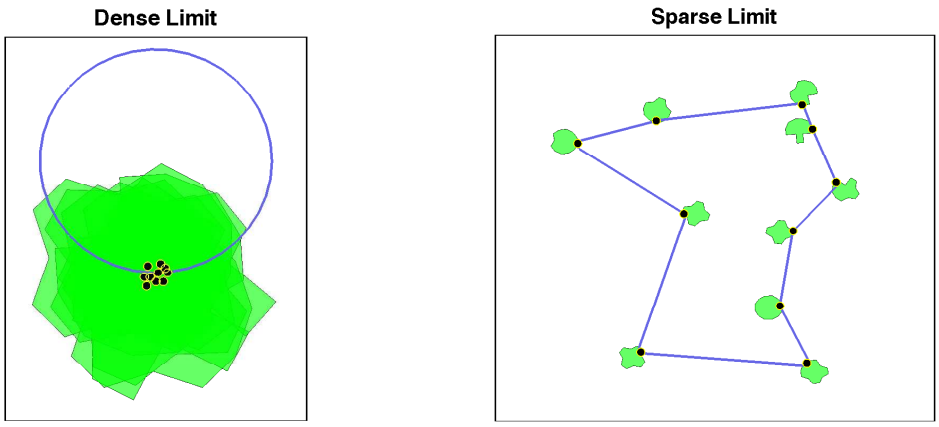



Figure 5.3. In the *dense limit* (left) as the distances between targets are much smaller than the minimum turning radius, there can be a large penalty incurred ($\Omega(n)$, see Theorem 5.1 and proof) by solving the DTSP instead of the PVDTSP. In particular, if the densely packed targets are sufficiently noncolinear, an aircraft solving the PVDTSP can photograph all targets in a single pass (shown as blue circle), but an aircraft solving the DTSP would only be able to photograph two targets per pass, thus requiring a tour at least the length of $\frac{n}{2}$ minimum turn radius circles. In the *sparse limit* (right) when the minimum turning radius and visibility set diameters are much smaller than the distances between targets, there is no significant advantage to solving the PVDTSP over the DTSP nor over the ETSP.

### 5.2.1 Calculating Visibility Regions

In order to calculate the visibility region $\mathcal{V}(\mathcal{T}_i)$ of a target, it is necessary to know the target location and to have a computer model/representation of the terrain. This representation may be either a vector format, e.g., a TIN (Triangulated Irregular Network), or a raster format, e.g., a DEM (Digital Elevation Map)

such as the military's DTED (Digital Terrain Elevation Data). The necessary data to build a terrain model could be gathered, e.g., by LIDAR (LIght Detection And Ranging), SAR (Synthetic Aperture Radar), or photogrammetry. Once the terrain model has been built, the visibility region of a target may be calculated using a "sweeping algorithm" [70, 71, 74] in the vector case, or Bresenham's line algorithm [98] in the raster case.

## 5.3 Sampling-Based Roadmap Methods

In this section we present two sampling-based roadmap methods for the PVDTSP. These methods operate by sampling a finite discrete set of poses from the continuous Dubins state space in order to approximate the PVDTSP instance by an FOTSP instance, then applying an FOTSP algorithm. We call the approximating FOTSP instance a *PVDTSP roadmap*. In Sec. 5.3.1 we explain in detail how to construct a PVDTSP roadmap. The methods that use the roadmap are then described in Sec. 5.3.2 and 5.3.3. We provide a numerical study in Sec. 5.3.4 and in Sec. 5.3.5 describe the relationship between the sampling-based roadmap methods in the present work and those used in the robotics literature for collision-free path planning. Later in Sec. 5.5 we explain how our methods can be extended to handle wind, airspace constraints, any vehicle dynamics, and open-path problems.

### 5.3.1 Roadmap Construction

To define a PVDTSP roadmap we first need definitions of the ATSP (Asymmetric TSP) and FOTSP (Finite One-in-a-set TSP) which are more precise than those given in Sec. 5.1.

**Definition 5.1** (ATSP)**.** *Given a weighted directed graph $\mathcal{G} = (V, E)$ where $V$ is a finite set of vertices*

$$\{v_1, v_2, v_3, \ldots, v_{n_V}\}$$

*and $E$ a set of directed edges with weights*

$$\{w_{i,j} | i, j \in \{1, 2, 3, \ldots, n_V\} \text{ and } i \neq j\},$$

*the* Asymmetric TSP (ATSP) *is to find a directed cycle of minimum cost which visits every vertex in $V$ exactly once.*

**Definition 5.2** (FOTSP)**.** *Suppose we have a weighted directed graph $\mathcal{G} = (V, E)$ as in Def. 5.1, but now the vertices are partitioned into finitely many nonempty mutually exclusive vertex sets called* clusters *$S = \{S_1, S_2, S_3, \ldots, S_{n_S}\}$, so that the vertices can be written as*

$$V = \Big\{ v_{(1,1)}, v_{(1,2)}, v_{(1,3)}, \ldots, v_{(1,n_{S_1})}, v_{(2,1)}, v_{(2,2)}, v_{(2,3)}, \ldots, v_{(2,n_{S_2})}, \ldots \\ \ldots, v_{(n_S,1)}, v_{(n_S,2)}, v_{(n_s,3)}, \ldots, v_{(n_S,n_{S_{n_S}})} \Big\}. \tag{5.5}$$

*Then the* Finite One-in-a-set TSP (FOTSP) *is to Find a directed cycle of minimum cost which visits at least one vertex from each cluster.*

**Definition 5.3** (PVDTSP Roadmap)**.** *A* roadmap *for a PVDTSP instance is an FOTSP instance, as per Def. 5.2, where there is one cluster for each polygon. The vertices $V$ are obtained by sampling a finite set of poses in each polygon and assigning them to the respective cluster. The edges $E$ are obtained by making all possible inter-cluster connections using Dubins minimum time state-to-state distances as weights.*

We perform the pose sampling in Def. 5.3 using what is known as a *quasirandom sequence*, although it could also be performed with a random sequence, uniform grid, or some heuristic. A quasirandom sequence is a deterministic sequence which densely fills a space and concurrently optimizes a generalized notion of resolution such as *dispersion*.[10]. Given a set of samples in a metric space, the

[10] There exist other generalized notions of resolution in the literature, e.g., *discrepancy* is usually used in the Monte-Carlo integration context [99]

dispersion of that set is the radius of the largest empty ball. Although there are many different quasirandom sequences to choose from in the literature [99, 18, 62], we have chosen to use *Halton* sequences for simplicity, efficiency, and because they (1) are asymptotically optimal with respect to dispersion, (2) have, with high probability, better dispersion than uniform random sampling, and (3) allow more flexibility in the number of samples than a regular grid. Halton sequences are defined formally as follows.

**Definition 5.4** (Halton Sequence [100])**.** *Let $b_1, \ldots, b_s$ be coprime positive integers greater than 1. For each $j \in \{1, \ldots, s\}$, let the base $b_j$ representation of an integer $k$ be given by*

$$k = \sum_i a_{ij} b_j^i \qquad (a_{ij} \in \{0, 1, \ldots, b_j - 1\}).$$

*Let*

$$\Phi_{b_j}(k) := \sum_i a_{ij} b_j^{-(i+1)}.$$

*Then the* $s$-dimensional Halton sequence $h_{(b_1,\ldots,b_{s-1})}(k) : \mathbb{N} \to [0,1]^s$ *is*

$$h_{(b_1,\ldots,b_{s-1})}(k) = (\Phi_{b_1}(k), \Phi_{b_2}(k), \ldots, \Phi_{b_d}(k)).$$

A Halton sequence produces samples on the $s$-dimensional unit box in $\mathbb{R}^s$, for some $s$, so in order to use it for sampling tours, we must show how to map samples from a unit box onto poses in the polygons of a PVDTSP instance. The definitions and main convergence result Theorem 5.2 to follow show that, without loss of generality, we may construct our PVDTSP roadmaps by sampling Halton points on a 2-dimensional unit box even though the full Dubins state space SE(2) is 3-dimensional. In particular, it suffices to map Halton points on the 2-dimensional unit box to *entry poses* of the polygons as shown in Fig. 5.3.1. By *entry pose of a polygon* we mean a pose which is positioned on the polygon's perimeter and either oriented towards the polygon's interior or parallel with its boundary.

**Definition 5.5** (Metric Space of Entry Poses $(X_{\text{entry}}, \rho_X)$)**.** *Let* $X_{\text{entry}} \subset X = \text{SE}(2)$ *be the 2-dimensional set of all Dubins states which correspond to an entry pose of some polygon in a PVDTSP instance. We endow* $X_{\text{entry}}$ *with the metric* $\rho_X$ *as follows. For any two points* $\mathbf{x} = (x, y, \psi)$ *and* $\mathbf{x}' = (x', y', \psi')$ *in* $X_{\text{entry}}$*,*

$$\rho_X(\mathbf{x}, \mathbf{x}') := \underbrace{\sqrt{|x - x'|^2 + |y - y'|^2}}_{\text{Euclidean metric on } \mathbb{R}^2} + \underbrace{\min\{|\psi - \psi'|, 2\pi - |\psi - \psi'|\}}_{\text{geodesic distance on } \mathbb{S}}. \quad (5.6)$$

To achieve a prescribed dispersion on a bounded $s$-dimensional continuous space can require arbitrarily many more samples than to achieve the same dispersion on a codimension one surface. Constructing a PVDTSP roadmap by sampling on the 2-dimensional $X_{\text{entry}}$ instead of the 3-dimensional $X$ should therefore significantly reduce the computational time complexity of any algorithm which uses the roadmap.

**Definition 5.6** (Halton Entry Pose Induced PVDTSP Roadmap $\mathcal{R}_i$)**.** *Let* $\mathcal{R}_i$ *denote a PVDTSP roadmap, as per Def. 5.3, where the vertices are obtained from mapping the first* $i$ *terms of the 2-dimensional Halton quasirandom sequence of bases 2 and 3 onto the space of entry poses as in Fig. 5.3.1.*

**Definition 5.7** (Metric Space of Dubins Tours $(\mathcal{D}, \rho_{\mathcal{D}})$)**.** *Let* $\mathcal{D}$ *be the set of all finite-length closed Dubins tours in the plane. We endow* $\mathcal{D}$ *with a metric* $\rho_{\mathcal{D}}$ *as follows. For any two tours* $\tau : \mathbb{S} \to \mathbb{R}^2$ *and* $\tau' : \mathbb{S} \to \mathbb{R}^2$ *in* $\mathcal{D}$*,*

$$\rho_{\mathcal{D}}(\tau, \tau') := \inf_{f:\mathbb{S}\leftrightarrow\mathbb{S}} \sup_{t\in\mathbb{S}} ||\tau(t) - \tau'(f(t))||_2, \quad (5.7)$$

*where* $\inf_{f:\mathbb{S}\leftrightarrow\mathbb{S}}$ *is the infimum over all reparameterizations between the tours,* $\sup_{t\in\mathbb{S}}$ *is the supremum over all positions along the tours, and* $||\ ||_2$ *is the* $L_2$ *norm in* $\mathbb{R}^2$*.*

**Definition 5.8** (Set of PVDTSP-feasible Dubins Tours $\mathcal{D}_{\text{feas}}$)**.** *Let* $\mathcal{D}_{\text{feas}}$ *be the set of all tours in* $\mathcal{D}$ *which pass through every polygon of a PVDTSP instance.*

We are now ready to state the main convergence result which will directly lead to convergence properties of the methods in Sec. 5.3.2 and 5.3.3.

**Theorem 5.2** (Roadmap Convergence)**.** *Let $\{\tau_i\}_{i=1}^{\infty}$ be the sequence of best tours contained in the sequence of roadmaps $\{\mathcal{R}_i\}_{i=1}^{\infty}$, respectively. Then the sequence of costs $\{C(\tau_i)\}_{i=1}^{\infty}$ is nonincreasing and*

$$\lim_{i\to\infty} C(\tau_i) \leq \inf_{\tau\in \operatorname{int}\mathcal{D}_{\text{feas}}} C(\tau). \tag{5.8}$$

*Proof.* From Def. 5.6 we know that a roadmap $\mathcal{R}_i$ is contained in another roadmap $\mathcal{R}_j$ whenever $j \geq i$, therefore the best tour in $\mathcal{R}_i$ is also in $\mathcal{R}_j$. This ensures the sequence of costs $\{C(\tau_i)\}_{i=1}^{\infty}$ is nonincreasing. The limit of the sequence of costs on the left hand side of Eq. 5.8 must exist because it is monotonic and lower bounded by zero. To prove the inequality it suffices to show that for all $\epsilon > 0$ there exists $N$ such that $i > N$ implies

$$C(\tau_i) \leq \inf_{\tau\in \operatorname{int}\mathcal{D}_{\text{feas}}} C(\tau) + \epsilon. \tag{5.9}$$

By definition of infimum, there exists a sequence of tours $\{\tau'_j\}_{j=1}^{\infty}$ in $\operatorname{int}\mathcal{D}_{\text{feas}}$ such that

$$\lim_{j\to\infty} C(\tau'_j) = \inf_{\tau\in \operatorname{int}\mathcal{D}_{\text{feas}}} C(\tau),$$

i.e., for all $\epsilon > 0$ there exists $N_1$ such that $j > N_1$ implies

$$C(\tau'_j) \leq \inf_{\tau\in \operatorname{int}\mathcal{D}_{\text{feas}}} C(\tau) + \frac{\epsilon}{2}. \tag{5.10}$$

A tour $\tau'_j$ must first enter each polygon at a unique entry pose. Because we are sampling poses densely in $X_{\text{entry}}$, we can always choose $i$ large enough that $\mathcal{R}_i$ has a set of entry poses arbitrarily close to the entry poses of $\tau'_j$ (with respect to the metric $\rho_X$). This together with the fact that $\tau'_j$ is in the interior of $\mathcal{D}_{\text{feas}}$ implies that for all $\epsilon > 0$ there exists $N_2$ such that

$$C(\tau_i) \leq C(\tau'_j) + \frac{\epsilon}{2} \tag{5.11}$$

whenever $i > N_2$. Combining Eq. 5.10 and 5.11, we obtain the desired result that for all $\epsilon > 0$ there exists $N = \max\{N_1, N_2\}$ such that $i > N$ implies

$$\begin{aligned} C(\tau_i) &\leq C(\tau'_j) + \tfrac{\epsilon}{2} \\ &\leq \inf_{\tau\in \operatorname{int}\mathcal{D}_{\text{feas}}} C(\tau) + \tfrac{\epsilon}{2} + \tfrac{\epsilon}{2} \\ &= \inf_{\tau\in \operatorname{int}\mathcal{D}_{\text{feas}}} C(\tau) + \epsilon. \end{aligned}$$

□

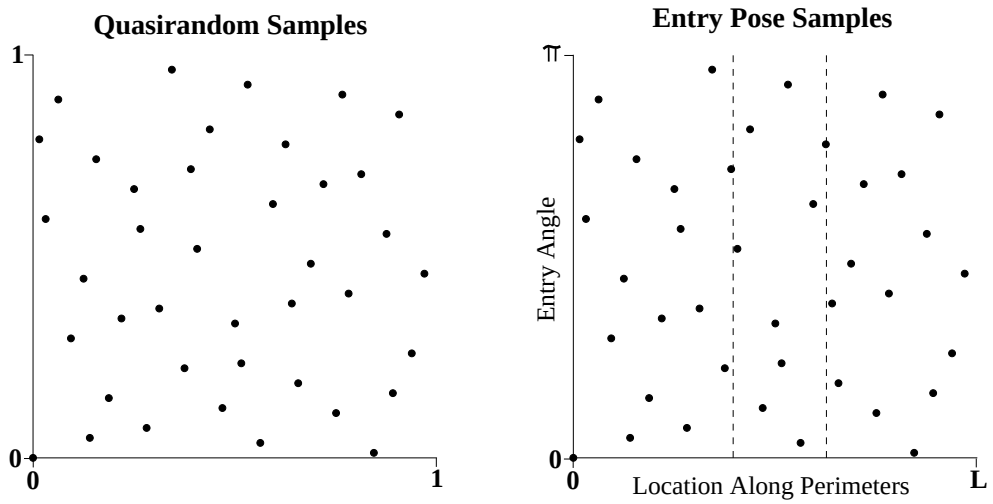



Figure 5.4. Green polygons are target visibility sets of a PVDTSP instance. The roadmap for a PVDTSP instance is an FOTSP instance which we construct in three steps. First we sample points from a Halton quasirandom sequence on the unit box (left). Second we map the quasirandom samples onto the space of entry poses represented by another box (right), where the horizontal axis represents a parameterization of position along the 1D polygon perimeters and the vertical axis represents entry angle in radians. These entry poses are the roadmap vertices and the dashed lines (right) show the separation between polygons. (continued)

In words, Theorem 5.2 states that the best tour cost taken from the sequence of (Halton entry pose induced) roadmaps is no greater, in the limit, than the cost of any nonisolated feasible tour. A roadmap may by chance contain an isolated global optimal tour, hence Eq. 5.8 is an inequality rather than equality. An example of an isolated global optimum is shown in Fig. 5.3.1.

### 5.3.2 Resolution Complete Method

We describe in this section a method which is *resolution complete*, which means it provably converges, in the limit as the number of samples in the roadmap

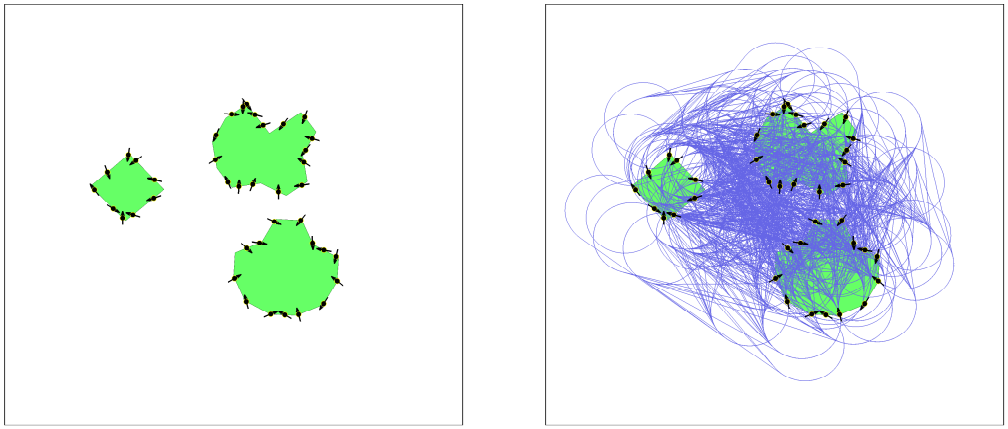


Figure 5.4. (continuation) The samples in the boxes correspond precisely to the entry pose samples shown on the plan view of the polygons (left). The vertices are partitioned into clusters according to which polygon they belong to. The third and last step is to create the roadmap edges by making all possible inter-cluster connections between vertices using Dubins shortest paths (right, blue curves). The edges are thus weighted by their Dubins distances. For comparison, an example roadmap for collision-free path planning is shown in Fig. 5.3.5.

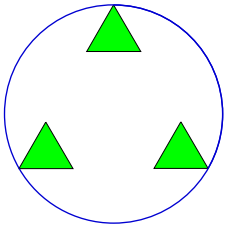


Figure 5.5. Isolated global optima can exist for a PVDTSP instance. In this example an isolated global optimal tour consists of the minimum turn radius circle (blue) just touching the outer vertices of the polygons.

increases, to a solution at least as good as any nonisolated solution.[11] A procedural

[11]This definition of resolution complete differs slightly from the useage in the context of sampling-based roadmap methods for collision free path planning, where it means that a method is guaranteed to find a nonisolated collision-free path as long as there are enough samples. However, the definitions are deliberately compatible so that a resolution complete collision-free path planner can be used in conjunction with our PVDTSP method in case we desire to solve

outline is shown in Table 5.1. The input consists of $n$ polygons, a vehicle minimum turn radius $r_{\min}$, and a sample count $n_{\text{samples}}$. First a roadmap is constructed by sampling $n_{\text{samples}}$ Halton entry poses as per Def. 5.6. Second, the roadmap FOTSP instance is converted to an ATSP instance using the *Noon-Bean transformation* [60], illustrated in Fig. 5.3.2 and defined as follows.

**Definition 5.9** (Noon-Bean Transformation)**.** *Suppose we are given an FOTSP instance specified, as in Def. 5.2, by a weighted directed graph $\mathcal{G} = (V, E)$ together with a partitioning into clusters $S = \{S_1, S_2, S_3, \ldots, S_{n_S}\}$. Then the* Noon-Bean Transformation *of this FOTSP instance is an ATSP instance $\mathcal{G}' = (V', E')$ constructed as follows. Begin with $\mathcal{G}' = \mathcal{G}$, i.e., let $V' = V$ and $E' = E$, then make these three modifications to $E'$:*

*(i) For each cluster, add zero-weight directed edges to $E'$ to create a zero-cost cycle which traverses all the vertices of the cluster (so there are a total of $n_S$ zero-cost cycles),*

*(ii) cyclically shift intercluster edges of $E'$ so that they emanate from the preceeding vertex in their respective zero-cost cycles, and*

*(iii) add a large penalty $M = \sum_{i,j} w_{i,j}$, i.e., the total of all weights in $\mathcal{G}$, to the weight of all interlcuster edges in $E'$.*

The third step of the method is to solve the ATSP instance, which can be done using any exact ATSP solver. The fourth and final step is to extract the PVDTSP

a PVDTSP with obstacles. We address these topics further in Sec. 5.3.5 and 5.5.1

Table 5.1. Outline of Resolution Complete Method for the PVDTSP

1: Construct roadmap by sampling entry poses on the polygon boundaries
2: Use Noon-Bean transform to convert Roadmap to an ATSP instance
3: Solve ATSP instance
4: Extract PVDTSP solution from ATSP solution

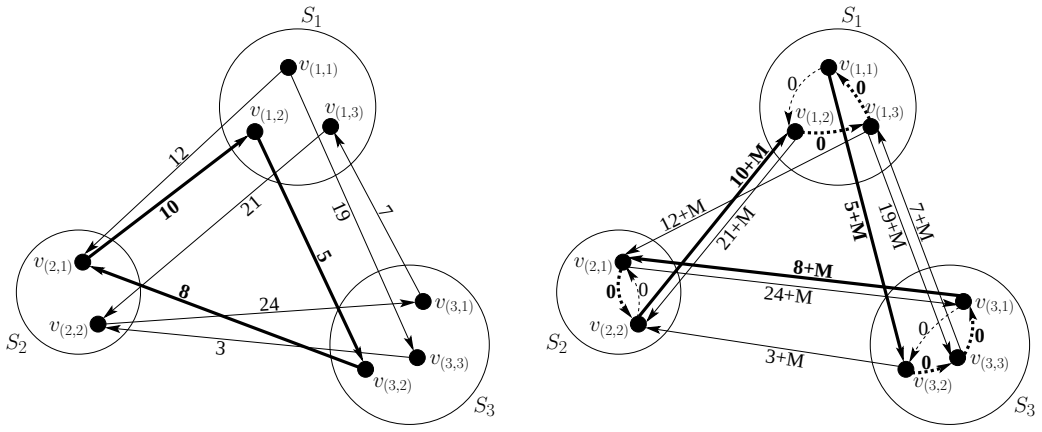



Figure 5.6. An FOTSP instance such as this example (left) can be transformed into an equivalent ATSP instance (right) using the Noon-Bean transformation. This transformation consists of (1) adding to each cluster a zero-cost cyle which traverses all the vertices of that cluster, (2) cyclically shifting intercluster edges so that they emanate from the preceeding vertex in their respective zero-cost cycles, and (3) adding a large penalty $M$ to interlcuster edges so that each cluster is only visited once. Once a solution to the ATSP instance is found, a solution to the FOTSP instance can by extracted by taking only the first vertex visited in each cluster, thus skipping fictitious zero-cost edges. Bold edges show equivalent optimal tours in the example FOTSP and ATSP instances.

solution form the ATSP solution by taking only the first vertex visited in each cluster, thus skipping the fictitious zero-cost edges.

The convergence of the method as the number of samples $n_{\text{samples}}$ goes to infinity is captured in the following corollary to Theorem 5.2.

**Corollary 5.1** (Convergence of Resolution Complete Method)**.** *Let $\{\tau_i\}_{i=1}^{\infty}$ be the sequence of tours computed by the resolution complete method when applied to the sequence of roadmaps $\{\mathcal{R}_i\}_{i=1}^{\infty}$, respectively. Then the sequence of costs $\{C(\tau_i)\}_{i=1}^{\infty}$ is nonincreasing and*

$$\lim_{i\to\infty} C(\tau_i) \leq \inf_{\tau\in\operatorname{int}\mathcal{D}} C(\tau). \tag{5.12}$$

*Proof.* It is proven in [60] that the Noon-Bean transformation is exact in the sense that if the exact solution of the ATSP instance is found, then the extracted solution to the FOTSP instance will also be exact. This means the method will always find the best tour in a given roadmap. The corollary thus follows directly from Theorem 5.2. □

Building the roadmap takes time complexity $\mathcal{O}(n_{\text{samples}}^2)$ because adding a vertex or edge costs constant time and there are $\mathcal{O}(n_{\text{samples}}^2)$ edges (no more than in a complete graph). The Noon-Bean transformation also takes time complexity $\mathcal{O}(n_{\text{samples}}^2)$ because it adds only $n_{\text{samples}}$ zero-weight edges, but modifies one at a time the other $\mathcal{O}(n_{\text{samples}}^2)$ edges. Solving the ATSP instance of $n_{\text{samples}}$ vertices is NP-hard and therefore we cannot expect to do it in guaranteed polynomial time. However, state-of-the-art heuristic ATSP solvers are effectively exact and have an (empirically determined) average case runtime of $\mathcal{O}(n_{\text{samples}}^{2.2})$ [101, 102]. We address further in Sec. 5.3.4 the issue of ATSP solver exactness versus effective exactness. Extracting the PVDTSP solution from the ATSP solution takes only $\mathcal{O}(n_{\text{samples}})$ time, so supposing we use a heuristic ATSP solver, we can expect the average case runtime of the entire method to be

$$\mathcal{O}(n_{\text{samples}}^{2.2}). \tag{5.13}$$

### 5.3.3 Approximate Dynamic Programming Method

The method we describe in this section uses approximate dynamic programming. A procedural outline is shown in Table 5.2. The input consists of $n$ polygons, a vehicle minimum turn radius $r_{\min}$, and a sample count $n_{\text{samples}}$. First a roadmap is constructed by sampling $n_{\text{samples}}$ Halton entry poses as per Def. 5.6. Second, the *FST (Fischetti-Salazar-Toth) transformation* [61], defined in Def. 5.10

Table 5.2. Outline of Approximate Dynamic Programming Method for the PVDTSP

| |
|---|
| 1: Construct roadmap by sampling entry poses on the polygon boundaries |
| 2: Use FST transform to obtain an ATSP instance from the roadmap |
| 3: Solve ATSP instance to obtain a cluster ordering |
| 4: Solve Dynamic Programs induced by cluster ordering from ATSP solution |
| 5: Select best Dynamic Program solution as PVDTSP solution |

and illustrated in Fig. 5.3.3, is applied to the roadmap FOTSP instance to obtain an ATSP instance.

**Definition 5.10** (FST Transformation)**.** *Suppose we are given an FOTSP instance specified, as in Def. 5.2, by a weighted directed graph* $\mathcal{G} = (V, E)$ *together with a partitioning into clusters* $S = \{S_1, S_2, S_3, \ldots, S_{n_S}\}$*. Then the* FST Transformation *of this FOTSP instance is an ATSP instance* $\mathcal{G}' = (V', E')$ *constructed as follows. There is one vertex in* $V'$ *for every cluster of the FOTSP instance. There is a directed edge from vertex* $v'_i \in V'$ *to vertex* $v'_j \in V'$ *if and only if there exists a directed edge from cluster* $S_i$ *to cluster* $S_j$*. The weight* $w'_{i,j}$ *of the directed edge from each such* $v'_i$ *to* $v'_j$ *is the arithmetic mean of the weights of all directed edges from* $S_i$ *to* $S_j$*.*

Solving the ATSP instance in the third step gives an approximate order $p$ to visit the roadmap FOTSP clusters. From this point on, any edges in the roadmap which do not satisfy order $p$ are ignored. Making use of the cluster and vertex labeling scheme introduced in Eq. 5.5, we now describe the fourth step of the method. Since we can perform a relabeling as necessary, suppose without loss of generality that $p = (1, 2, 3, \ldots, n_S)$, i.e., the approximate cluster ordering is $S_1, S_2, S_3, \ldots, S_{n_S}$. Suppose further that we known the optimal roadmap FOTSP solution passes through the $j^*$th vertex $v_{(1,j^*)}$ of the first cluster. Treating clusters

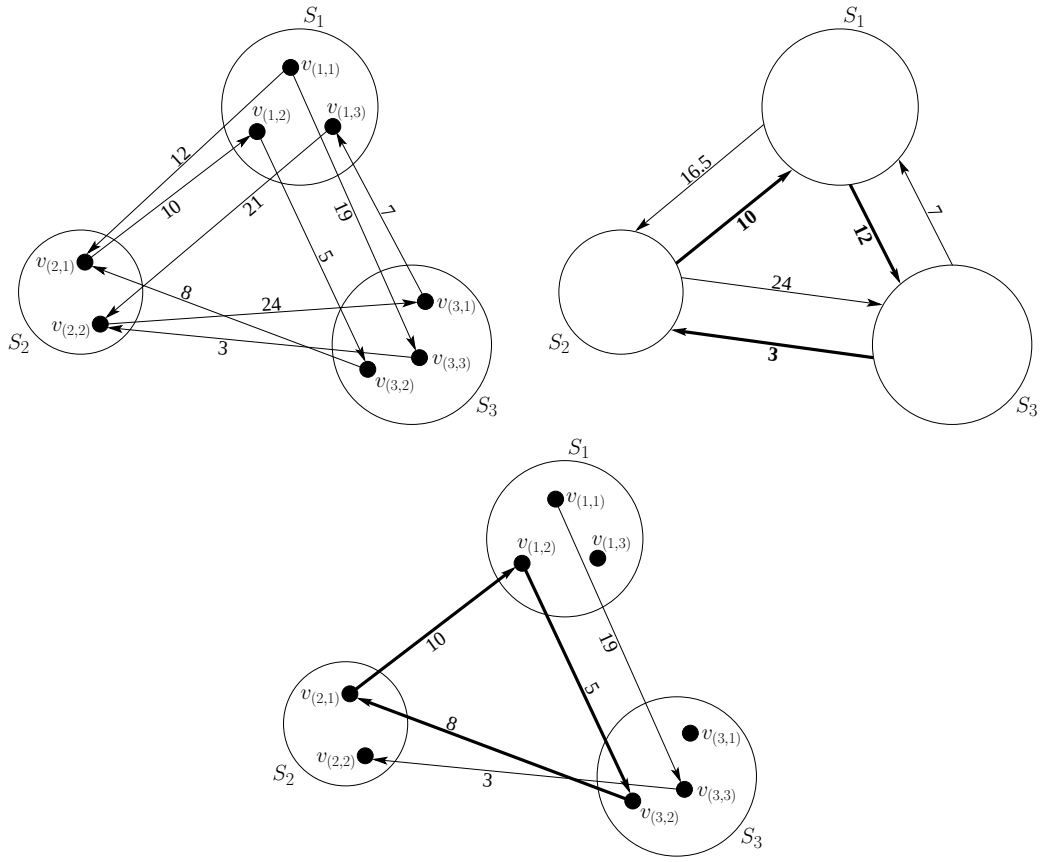



Figure 5.7. The FST transform of an FOTSP instance (upper left) is an ATSP instance (upper right) where (1) the vertices of the ATSP instance correspond to clusters of the FOTSP instance, and (2) the weight of each directed edge in the ATSP instance is the average weight of directed edges between the respective clusters in the FOTSP instance. Solving the ATSP instance gives an approximate ordering, say $p$, of the clusters of the FOTSP instance. If one ignores in the FOTSP instance all edges not satisfying the ordering $p$ (bottom), then an approximate FOTSP solution (bottom bold) can be found by dynamic programming with the clusters as stages.

as stages, an optimal FOTSP solution satsifying order $p$ can be found by solving the dynamic programming recursion

$$
\begin{aligned}
G^*(v_{(n_S,j)}) &= d(v_{(n_S,j)}, v_{(1,j^*)}) \quad (j = 1, \ldots, n_{S_{n_S}}) \\
G^*(v_{(i,j)}) &= \min_{k \in \{1,\ldots,n_{S_{i+1}}\}} \Big\{ d(v_{(i,j)}, v_{(i+1,k)}) + G^*(v_{(i+1,k)}) \Big\} \quad (i = n_S, \ldots, 2; j = 1, \ldots, n_{S_i}) \\
G^*(v_{(1,j^*)}) &= \min_{k \in \{1,\ldots,n_{S_2}\}} \Big\{ d(v_{(1,j^*)}, v_{(2,k)}) + G^*(v_{(2,k)}) \Big\},
\end{aligned}
$$

where $G^*(v)$ denotes the optimal-cost-to-go from a vertex $v$, and $d(v, v')$ is the weight of the directed edge from vertex $v$ to vertex $v'$. Since it is not known a prior which vertex in the first cluster is $i^*$, one dynamic program must be solved for each vertex in the first cluster, hence step five of the method is to select the best out of $n_{S_1}$ dynamic program solutions.

The convergence of the method as the number of samples $n_{\text{samples}}$ goes to infinity is captured in the following corollary to Theorem 5.2.

**Corollary 5.2** (Convergence of Approximate Dynamic Programming Method)**.** *Let $\{\tau_i\}_{i=1}^{\infty}$ be a sequence of tours computed by the approximate dynamic programming method when applied to the sequence of roadmaps $\{\mathcal{R}_i\}_{i=1}^{\infty}$, respectively. If there exists $N$ such that $\tau_i$ satisfies an order $p$ for all $i > N$ then the sequence of costs $\{C(\tau_i)\}_{i=N+1}^{\infty}$ is nonincreasing and*

$$
\lim_{i \to \infty} C(\tau_i) \leq \inf_{\tau \in (\operatorname{int} \mathcal{D}_{\text{feas}})_p} C(\tau), \tag{5.14}
$$

*where $(\mathcal{D}_{\text{feas}})_p$ is the set of all PVDTSP-feasible tours which satisfy order $p$.*

*Proof.* Let $\{(\mathcal{R}_i)_p\}_{i=1}^{\infty}$ be the sequence of roadmaps as in Def. 5.6, but where only edges satisfying order $p$ are allowed. Dynamic programming will always find the best tours in these $p$-limited roadmaps, therefore the proof of this corollary is exactly the same as the proof for Theorem 5.2, except $\mathcal{D}_{rmfeas}$ is replaced by $(\mathcal{D}_{\text{feas}})_p$ and $\{\mathcal{R}_i\}_{i=1}^{\infty}$ by $\{(\mathcal{R}_i)_p\}_{i=1}^{\infty}$. □

Building the roadmap takes time complexity $\mathcal{O}(n_{\text{samples}}^2)$ because adding a vertex or edge takes constant time and there are $\mathcal{O}(n_{\text{samples}}^2)$ edges (no more than in a complete graph). The FST transformation also takes time complexity $\mathcal{O}(n_{\text{samples}}^2)$ because it must access each of the $\mathcal{O}(n_{\text{samples}}^2)$ edges of the FOTSP instance in

order compute the weights of the edges in the ATSP instance. Since the ATSP is NP-hard, we assume a state-of-the-art heuristic ATSP solver with (empirically determined) average case runtime of $\mathcal{O}(n^{2.2})$ will be used [101, 102]. On average there will be $\frac{n_{\text{samples}}}{n}$ vertices per cluster, so the average case total time complexity of solving the dynamic programs is $\mathcal{O}(\frac{n_{\text{samples}}^2}{n})$. Adding up all the time complexities, we can expect the average case runtime of the entire method to be

$$\mathcal{O}(n^{2.2} + \frac{n_{\text{samples}}^2}{n} + n_{\text{samples}}). \tag{5.15}$$

### 5.3.4 Numerical Study

We have implemented the sampling-based roadmap methods of Sec. 5.3.2 and 5.3.3 in C++ on a 2.33 GHz i686. For solving ATSP instances, our implementations call the powerful LKH [102] solver as a subroutine. Strictly speaking, LKH is based on what is known as the Lin-Kernighan heuristic, and therefore is an inexact solver, i.e., it is not guaranteed to find the global optimal solution to an ATSP instance. Using an inexact ATSP solver with what we have been calling the "resolution complete method" means that it is no longer truly resolution complete and therefore not guaranteed to converge to a nonisolated global optimum. However, allowing ourselves a slight abuse of terminology, we retain the name "resolution complete method" in the presentation of our numerical results because state-of-the-art heuristic TSP solvers, e.g., LKH or Linkern, perform so well in practice that they are widely accepted as *effectively exact* for ATSP instances having up to hundreds or even thousands of nodes [102, 103]. Moreover, exact ATSP solvers can be extremely slow, sometimes taking hours to find a solution that an inexact

heuristic solver finds in only seconds. [12] Hours of computation time may not be available in UAV applications requiring online solutions.

Out of several dozen problem instances we experimented with, the results from three representative examples are shown in Table 5.3, Fig. 5.3.4, Fig. 5.3.4, and Fig. 5.3.4. In all examples the aircraft minimum turn radius was $r_{\min} = 3$ m. Both methods deliver good solutions and are suitably fast for online purposes when applied to PVDTSP instances having up to about 20 targets. The computation times for the approximate dynamic programming method are generally a little shorter than for the resolution complete method, but the resulting tours are also a little longer. The plots of computation time vs. sample count seem to match the predicted average case time complexities in Eq. 5.13 and 5.15. One can see, from the plots of solution quality vs. sample count and computation time vs. sample count, that a user of either method can indirectly trade off computation time for solution quality by adjusting the number of samples. The resolution complete method appears to monotonically converge to nonisolated global optima as the number of samples grows, so we presume the LKH solver is indeed effectively exact for these examples having up to 20 targets and 1500 samples. In just a few examples out of dozens we tested did we observe slight nonmonotonicity. This mostly occured when there were greater than 20 targets and 1500 samples. Although this indicates LKH is no longer effectively exact for the larger size problem instances, the approximate solutions it gave were consistently very good.

The PVDTSP instances used for experimentation in this section are the same as those used for testing the genetic algorithm presented in Sec. 5.4. In particular, the PVDTSP instances in Figures 5.3.4, 5.3.4, 5.3.4 correspond to those in Figures

[12] According to [102], the empirically determined average case run-time of LKH on an ATSP instance with $n_{\text{nodes}}$ nodes is $\mathcal{O}(n_{\text{nodes}}^{2.2})$.

5.4.3, 5.4.3, 5.4.3, respectively. For small instances with around 5 targets or less, the performance of the genetic algorithm, in terms of solution quality per computation time, is comparable to that of the sampling-based roadmap methods. For larger problem instances with greater than 5 targets the sampling-based roadmap methods perform significantly better.

### 5.3.5 Relationship to Methods for Collision-Free Path Planning

As mentioned in Sec. 5.1, sampling-based roadmap methods in the robotics literature, surveyed nicely in the recent texts [18] and [62], have traditionally been used exclusively for planning collision-free paths through continuous spaces by discretizing the obstacle-free portion of the space into a finite directed graph called a *roadmap*, e.g., as shown in Fig. 5.3.5. The roadmap can then be searched using standard shortest path algorithms such as Dijkstra or A$^*$ [104]. It is interesting to note that for collision-free path planning the roadmap vertices must be sampled from the full 3-dimensional Dubins state space, yet for a PVDTSP roadmap it is sufficient to sample only on the 2-dimensional space of entry poses (Fig. 5.3.1 vs.

Table 5.3. Statistics from resolution complete and approximate dynamic programming algorithms implemented in C++ on a 2.33 GHz i686.

| Instance | No. of Samples | Resolution Complete | | Approx. Dynamic Programming | |
|---|---|---|---|---|---|
| | | Computation Time | Tour Length | Computation Time | Tour Length |
| Fig. 5.3.4 5 targets | 400 | 8.05 s | 37.64 m | 6.12 s | 37.64 m |
| Fig. 5.3.4 10 targets | 800 | 53.54 s | 67.44 m | 51.42 s | 69.89 m |
| Fig. 5.3.4 20 targets | 1500 | 506.07 s | 118.99 m | 447.14 s | 142.03 m |

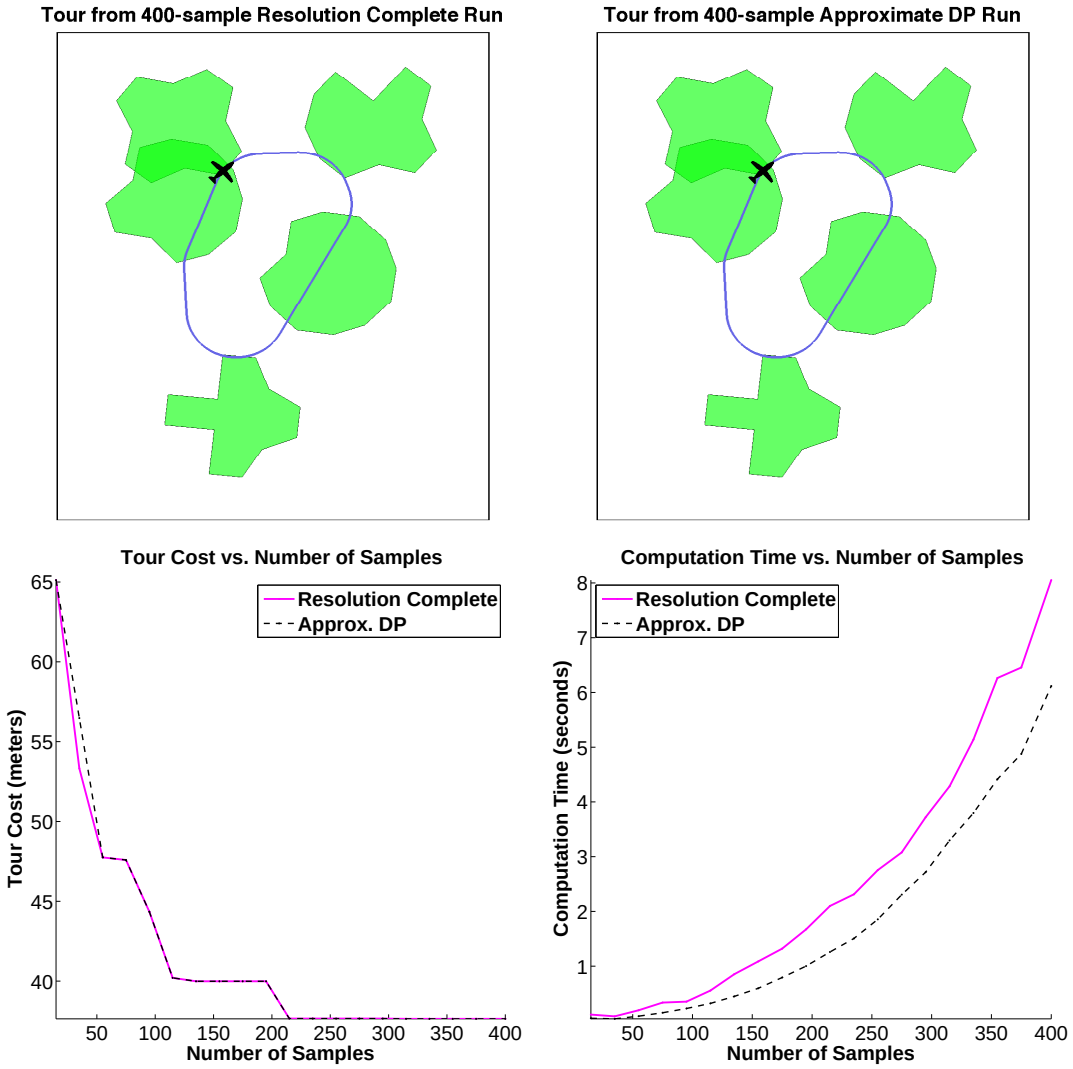



Figure 5.8. Computed example with $n = 5$ targets, aircraft minimum turn radius $r_{\min} = 3$ m. Green polygons represent the target visibility regions. Black dots are the tour nodes. Cf Table 5.3.

Fig. 5.3.5).

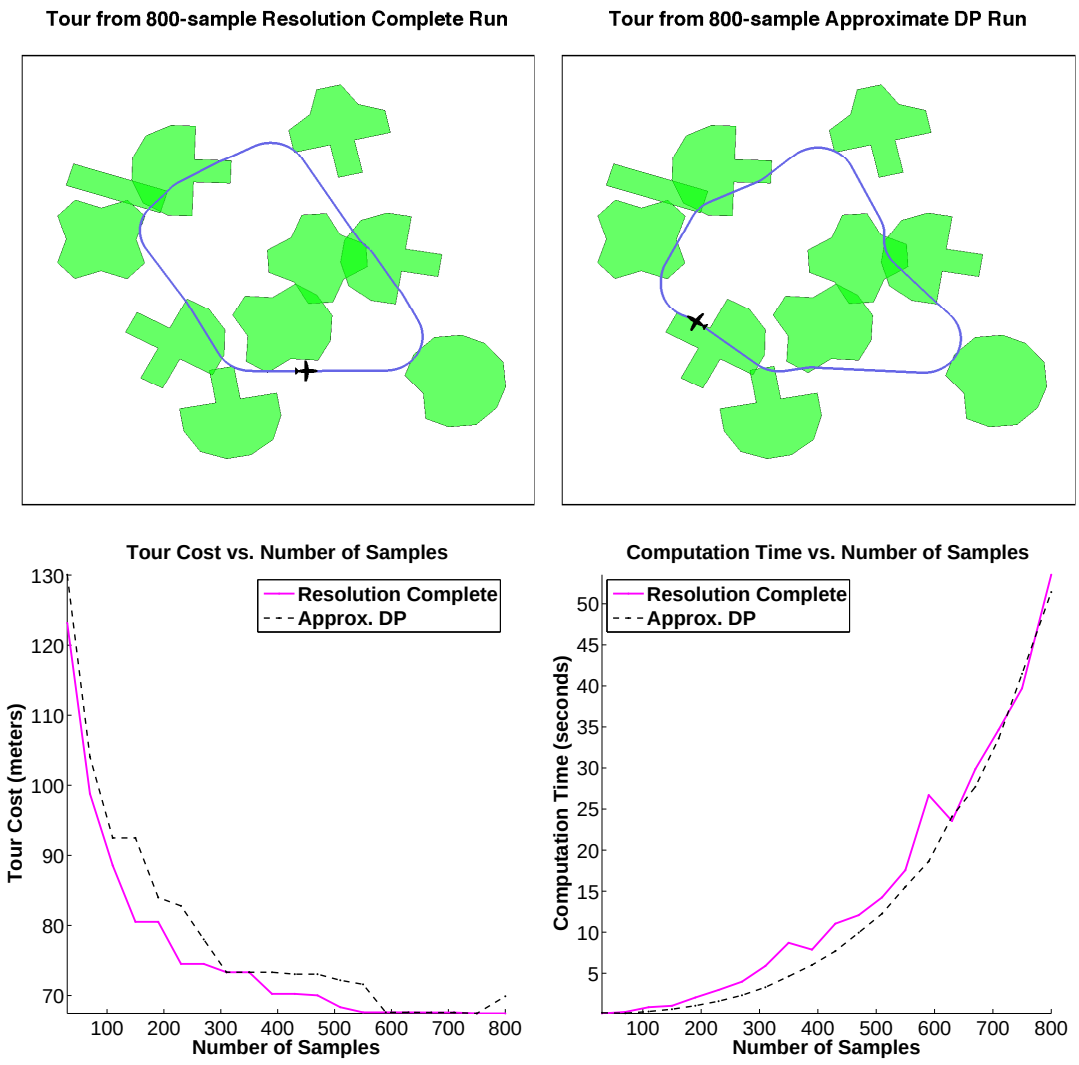



Figure 5.9. Computed example with $n = 10$ targets, aircraft minimum turn radius $r_{\min} = 3$ m. Green polygons represent target visibility regions. Black dots are the tour nodes. Cf Table 5.3.

## 5.4 A Genetic Algorithm

In this section we present a genetic algorithm which is easy to implement and delivers quick feasible solutions to the PVDTSP with monotonic improvement over runtime. Although there are no performance guarantees, we validate the

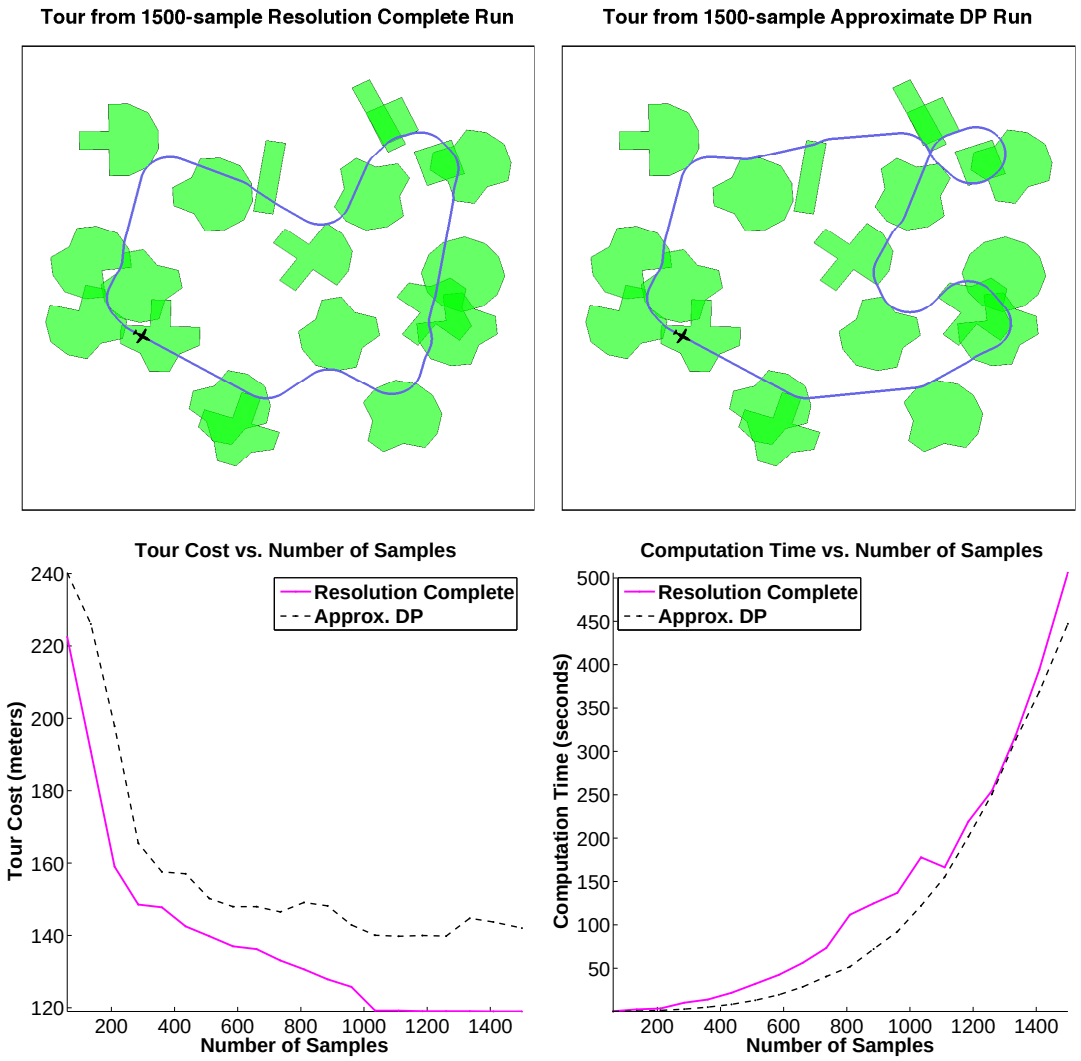



Figure 5.10. Computed example with $n = 20$ targets, aircraft minimum turn radius $r_{\min} = 3$ m. Green polygons represent target visibility regions. Black dots are the tour nodes. Cf Table 5.3.

algorithm with a Monte-Carlo numerical study in Sec. 5.4.3. Later in Sec. 5.5 we show that the algorithm can be extended to handle wind, airspace constraints, any vehicle dynamics, and open-path problems. For details on genetic algorithms in general, we suggest [105] and [64].

The first step in designing a genetic algorithm is to decide on an encoding to

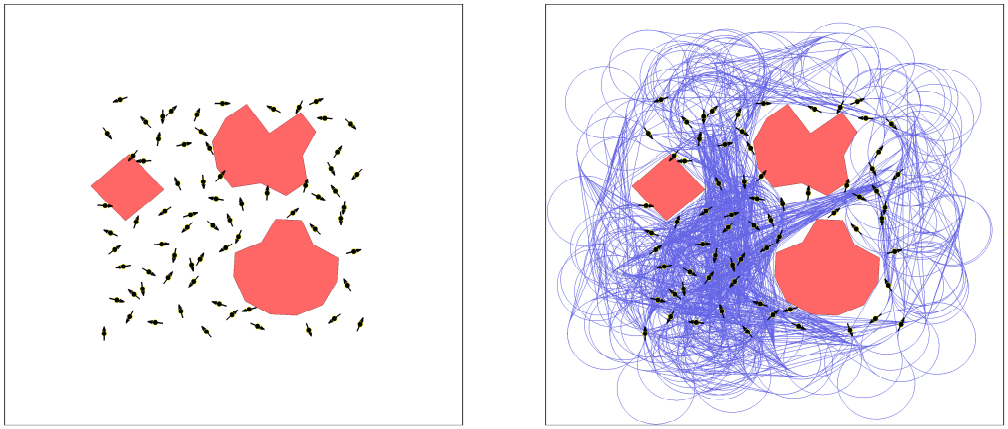


Figure 5.11. Suppose the red polygons are obstacles. A roadmap for computing collision-free Dubins paths between pairs of poses is a finite directed graph whose vertices are poses sampled from the freespace (black arrows, left) and whose edges are obtained by attempting to make collision-free connections between samples using a *local planning method*, usually a Boundary Value Problem solver (blue curves, right). In this example poses were sampled using a Halton quasirandom sequence in SE(2), however, one could alternatively use random or uniform grid sampling. To find a collision-free path between two poses, those poses are connected to the roadmap using the local planning method, then the roadmap is searched using a shortest path algorithm such as Dijkstra or A$^*$. Cf. Fig. 5.3.1.

represent candidate solutions. Each encoded solution will be a *chromosome* in the *population* which evolves over the course of the genetic algorithm. For our encoding we use a sequence of doubles

$$((\mathcal{T}_{k_1}, \mathbf{x}_1), (\mathcal{T}_{k_2}, \mathbf{x}_2), (\mathcal{T}_{k_3}, \mathbf{x}_3), \ldots, (\mathcal{T}_{k_n}, \mathbf{x}_n)), \tag{5.16}$$

where we refer to a double $(\mathcal{T}_{k_i}, \mathbf{x}_i)$ as a *node* of the tour. The sequence $k_1, \ldots, k_n$ represents a permutation of the target identifiers $1, \ldots, n$. This is the order in which the aircraft will visit the targets. Each aircraft state $\mathbf{x}_i$ is in the $k_i$th target's visibility set $\mathcal{V}(\mathcal{T}_{k_i})$. If a solution is accepted, the aircraft flies to each

state $\mathbf{x}_1, \ldots, \mathbf{x}_n$ in the order they appear in the chromosome, photographing $\mathcal{T}_{k_i}$ when it reaches $\mathbf{x}_i$. Shortest Dubins paths are flown between successive states in the sequence. We define the *fitness* of a chromosome $c$ to be $f(c) = 1/C(c)$, where $C$ is the cost shown in Eq. 5.2. We say a chromosome $c$ is more *fit* than another $c'$ if $f(c) > f(c')$.

Referring to the pseudocode in Table 5.4, we now describe how our genetic algorithm operates. The algorithm has five fixed parameters: *population size* $N_P$, *crossover probability* $p_x$, *mutation probability* $p_m$, *elite group size* $N_e$, and *number of generations* $N_g$. The population $P$ is initialized to a set of $N_P$ random chromosomes. A random chromosome is produced by randomly shuffling the integers $1, \ldots, n$, uniform randomly sampling points in the the targets' visibility sets (for the state positions), and uniform randomly sampling angles on the interval $[0, 2\pi)$ (for state orientations). Now we enter the main loop (line 2). At the beginning of the main loop, the $N_e$ fittest chromosomes are copied into the next generation population $P'$. This ensures the fitness of the fittest chomosome in $P$ can never decrease during the execution of the algorithm. Next, in order to make $P'$ have size $N_P$, we need to produce $N_P - N_e - 1$ offspring. Offspring are produced two at a time as follows. Two chromosomes, $c_{\text{mom}}$ and $c_{\text{dad}}$ are selected randomly from $P$ with probability proportional to their respective fitnesses. This is known as *roulette wheel* parent selection. With probability $p_x$, children $c_i$ and $c_{i+1}$ are generated by crossing over the parents, otherwise they are just copies of the parents (see Sec. 5.4.1 below for a detailed description of our crossover operation). Once the children are in place, we randomly perform mutation operators on them with probabilities $p_m$, $p_m$, $p_m/7$, and $p_m/7$, respectively[13] (see Sec. 5.4.2 below for a

[13]The last two mutation probabilities were scaled by 1/7 because it seems to give better performance.

detailed description of our mutation operators). Once all the children have been constructed and $P'$ has size $n$, $P$ and $P'$ are swaped, i.e. the new generation replaces the old. The process continues for $N_g$ generations. After termination, the algorithm returns the fittest chromosome in the population, which represents the shortest aircraft tour found.

### 5.4.1 Crossover

Our crossover operator, which appears in line 6 of the pseudocode Table 5.4, produces offspring by preserving partial tours from the parents. We have adapted for our problem the so-called Order Crossover (OX) [64] used for the TSP. An offspring is constructed by choosing a node subsequence from one parent, then filling in the remaining nodes in the order they appear in the other parent. We illustrate by an example having $n = 9$ targets. Let the parents be

$$c_{\text{mom}} = ((\mathcal{T}_2, \mathbf{x}_1), (\mathcal{T}_3, \mathbf{x}_2), (\mathcal{T}_5, \mathbf{x}_3), (\mathcal{T}_7, \mathbf{x}_4), (\mathcal{T}_4, \mathbf{x}_5), (\mathcal{T}_6, \mathbf{x}_6), (\mathcal{T}_9, \mathbf{x}_7), (\mathcal{T}_8, \mathbf{x}_8), (\mathcal{T}_1, \mathbf{x}_9))$$

$$c_{\text{dad}} = ((\mathcal{T}_6, \tilde{\mathbf{x}}_1), (\mathcal{T}_1, \tilde{\mathbf{x}}_2), (\mathcal{T}_7, \tilde{\mathbf{x}}_3), (\mathcal{T}_9, \tilde{\mathbf{x}}_4), (\mathcal{T}_3, \tilde{\mathbf{x}}_5), (\mathcal{T}_5, \tilde{\mathbf{x}}_6), (\mathcal{T}_8, \tilde{\mathbf{x}}_7), (\mathcal{T}_4, \tilde{\mathbf{x}}_8), (\mathcal{T}_2, \tilde{\mathbf{x}}_9)).$$

First, two cut points are uniform randomly chosen, represented by the bars $|$,

$$c_{\text{mom}} = ((\mathcal{T}_2, \mathbf{x}_1), (\mathcal{T}_3, \mathbf{x}_2) \mid (\mathcal{T}_5, \mathbf{x}_3), (\mathcal{T}_7, \mathbf{x}_4), (\mathcal{T}_4, \mathbf{x}_5), (\mathcal{T}_6, \mathbf{x}_6) \mid (\mathcal{T}_9, \mathbf{x}_7), (\mathcal{T}_8, \mathbf{x}_8), (\mathcal{T}_1, \mathbf{x}_9))$$

$$c_{\text{dad}} = ((\mathcal{T}_6, \tilde{\mathbf{x}}_1), (\mathcal{T}_1, \tilde{\mathbf{x}}_2) \mid (\mathcal{T}_7, \tilde{\mathbf{x}}_3), (\mathcal{T}_9, \tilde{\mathbf{x}}_4), (\mathcal{T}_3, \tilde{\mathbf{x}}_5), (\mathcal{T}_5, \tilde{\mathbf{x}}_6) \mid (\mathcal{T}_8, \tilde{\mathbf{x}}_7), (\mathcal{T}_4, \tilde{\mathbf{x}}_8), (\mathcal{T}_2, \tilde{\mathbf{x}}_9)).$$

Next, the sections between the cut points are placed directly into the offspring,

$$c_i = (\ -\ ,\ -\ \mid (\mathcal{T}_5, \mathbf{x}_3), (\mathcal{T}_7, \mathbf{x}_4), (\mathcal{T}_4, \mathbf{x}_5), (\mathcal{T}_6, \mathbf{x}_6) \mid -\ ,\ -\ ,\ -\ )$$

$$c_{i+1} = (\ -\ ,\ -\ \mid (\mathcal{T}_7, \tilde{\mathbf{x}}_3), (\mathcal{T}_9, \tilde{\mathbf{x}}_4), (\mathcal{T}_3, \tilde{\mathbf{x}}_5), (\mathcal{T}_5, \tilde{\mathbf{x}}_6) \mid -\ ,\ -\ ,\ -\ ).$$

It now remains to fill in the blank spaces $(-)$ in the offspring. The rest of $c_i$ is

Table 5.4. Genetic Algorithm for the PVDTSP

construct random initial population $P = \{c_1, c_2, \ldots, c_{N_c}\}$ of $N_P$ chromosomes;
**for all** generations $1, 2, 3, \ldots, N_g$ **do**
  copy $N_e$ best chromosomes from $P$ directly into new population $P'$;
  **for all** $i = 1, 3, 5, 7, \ldots, N_P - N_e - 1$ **do**
    roulette wheel select parents $c_{\text{mom}}$ and $c_{\text{dad}}$ from $P$;
    generate children $c'_i$ and $c'_{i+1}$ by crossover with probability $p_x$, otherwise $c'_i \leftarrow c_{\text{mom}}$ and $c'_{i+1} \leftarrow c_{\text{dad}}$;
    **for all** $j = i, i+1$ **do**
      orientation shift mutate $c'_j$ with probability $p_m$;
      position shift mutate $c'_j$ with probability $p_m$;
      swap mutate $c'_j$ with probability $p_m/7.0$;
      partial reverse mutate $c'_j$ with probability $p_m/7.0$;
    $P' = P' \cup \{c'_i, c'_{i+1}\}$;
  $P \leftarrow P'$; clear $P'$;
**return** best chromosome in $P$;

completed as follows. The order of nodes in $c_{\text{dad}}$, starting from the second cut point, is

$$((\mathcal{T}_8, \tilde{\mathbf{x}}_7), (\mathcal{T}_4, \tilde{\mathbf{x}}_8), (\mathcal{T}_2, \tilde{\mathbf{x}}_9), (\mathcal{T}_6, \tilde{\mathbf{x}}_1), (\mathcal{T}_1, \tilde{\mathbf{x}}_2), (\mathcal{T}_7, \tilde{\mathbf{x}}_3), (\mathcal{T}_9, \tilde{\mathbf{x}}_4), (\mathcal{T}_3, \tilde{\mathbf{x}}_5), (\mathcal{T}_5, \tilde{\mathbf{x}}_6)).$$

Deleting from this ordering the nodes with targets $\{\mathcal{T}_5, \mathcal{T}_7, \mathcal{T}_4, \mathcal{T}_6\}$ already present in the first offspring, we obtain

$$((\mathcal{T}_8, \tilde{\mathbf{x}}_7), (\mathcal{T}_2, \tilde{\mathbf{x}}_9), (\mathcal{T}_1, \tilde{\mathbf{x}}_2), (\mathcal{T}_9, \tilde{\mathbf{x}}_4), (\mathcal{T}_3, \tilde{\mathbf{x}}_5)).$$

Finally, this remaining sequence of nodes is inserted into the blank spaces of $c_i$, starting at the second cut point, to obtain

$$c_i \;=\; ((\mathcal{T}_9, \tilde{\mathbf{x}}_4), (\mathcal{T}_3, \tilde{\mathbf{x}}_5) \mid (\mathcal{T}_5, \mathbf{x}_3), (\mathcal{T}_7, \mathbf{x}_4), (\mathcal{T}_4, \mathbf{x}_5), (\mathcal{T}_6, \mathbf{x}_6) \mid (\mathcal{T}_8, \tilde{\mathbf{x}}_7), (\mathcal{T}_2, \tilde{\mathbf{x}}_9), (\mathcal{T}_1, \tilde{\mathbf{x}}_2)).$$

Similarly, the second offspring is

$$c_{i+1} \;=\; ((\mathcal{T}_4, \mathbf{x}_5), (\mathcal{T}_6, \mathbf{x}_6) \mid (\mathcal{T}_7, \tilde{\mathbf{x}}_3), (\mathcal{T}_9, \tilde{\mathbf{x}}_4), (\mathcal{T}_3, \tilde{\mathbf{x}}_5), (\mathcal{T}_5, \tilde{\mathbf{x}}_6) \mid (\mathcal{T}_8, \mathbf{x}_8), (\mathcal{T}_1, \mathbf{x}_2), (\mathcal{T}_2, \mathbf{x}_1)).$$

### 5.4.2 Mutation

Mutation, in lines 8-11 of the pseudocode Table 5.4, is used in case the initial population is not rich enough to find a good solution via crossover alone. We use four different kinds of mutation. In *orientation shift*, an index $i \in \{1, \ldots, n\}$ is chosen uniform randomly, then the aircraft azimuth $\psi$ within the state $\mathbf{x}_i$ is reset uniform randomly in the interval $[0, 2\pi)$ (radians). In *position shift*, an index $i \in \{1, \ldots, n\}$ is chosen uniform randomly, then the aircraft position $(x, y)$ within the state $\mathbf{x}_i$ is reset uniform randomly within the polygonal approximation of $\mathcal{V}(\mathcal{T}_{k_i})$. In *swap*, two indices $i, j \in \{1, \ldots, n\}$ are chosen uniform randomly, then the nodes $(\mathcal{T}_{k_i}, \mathbf{x}_i)$ and $(\mathcal{T}_{k_j}, \mathbf{x}_j)$ swap positions within the chromosome. In *partial reverse*, two indices $i, j \in \{1, \ldots, n\}$ are chosen uniform randomly, then the segment of the chromosome $(\mathcal{T}_{k_i}, \mathbf{x}_i), \ldots, (\mathcal{T}_{k_j}, \mathbf{x}_j)$ is reversed; this includes rotating the azimuth portion of those nodes' states by $\pi$ radians.

### 5.4.3 Numerical Study

We have implemented the genetic algorithm of Sec. 5.4 in C++ on a 2.33 GHz i686. For a baseline comparison, we also implemented the naive random search algorithm shown in Table 5.5. Out of several dozen problem instances we experimented with, the Monte-Carlo results from three representative examples are shown in Table 5.6, Fig. 5.4.3, Fig. 5.4.3, and Fig. 5.4.3. In all examples the aircraft minimum turn radius was $r_{\min} = 3$ m. The parameter values shown in Table 5.6 were the best of many different combinations of values we tested, so we presume they are close to optimal. The genetic algorithm delivered good

solutions suitably quickly for online purposes when applied to PVDTSP instances having up to about 20 targets. Tours produced by the genetic algorithm were on average about half the length of those produced by the pure random search. The monotonic improvement, seen in the plots of solution quality vs. computation time, illustrates how a user can directly trade off computation time for solution quality by stopping the algorithm anytime in order to obtain the best feasible solution so far.

The PVDTSP instances used for experimentation in this section are the same as those used for the sampling-based roadmap methods in Sec. 5.3. In particular, the PVDTSP instances in Figures 5.4.3, 5.4.3, 5.4.3 correspond to those in Figures 5.3.4, 5.3.4, 5.3.4, respectively. For small instances with around 5 targets or less, the performance of the genetic algorithm, in terms of solution quality per computation time, is comparable to that of the sampling-based roadmap methods. For larger problem instances with greater than 5 targets the sampling-based roadmap methods perform significantly better.

Table 5.5. Random Search Algorithm for the PVDTSP

```
1: c_best ← random feasible solution;
2: while t < T do
3:     c ← random feasible solution;
4:     if c is better than c_best then
5:         c_best ← c;
6: return c_best;
```

Table 5.6. Statistics from genetic and random search algorithms implemented in C++ on a 2.33 GHz i686.

| Instance | No. of Runs | Computation Time per Run | Genetic Algorithm Parameters | Genetic Algorithm Mean Best Tour Length | Random Search Mean Best Tour Length |
|---|---|---|---|---|---|
| Fig. 5.4.3 5 targets | 30 | 4.99 s | $N_P = 100$, $N_g = 500$, $N_e = 4$, $p_x = 0.7$, $p_m = 0.1$ | 36.99 m | 55.36 m |
| Fig. 5.4.3 10 targets | 30 | 20.66 s | $N_P = 100$, $N_g = 1000$, $N_e = 4$, $p_x = 0.7$, $p_m = 0.1$ | 77.33 m | 154.29 m |
| Fig. 5.4.3 20 targets | 30 | 87.90 s | $N_P = 100$, $N_g = 2000$, $N_e = 4$, $p_x = 0.7$, $p_m = 0.1$ | 173.85 m | 412.90 m |

## 5.5 Extensibility

### 5.5.1 Wind, Airspace Constraints, and Any Dynamics

In Sec. 5.2 we formulated the minimum time reconnaissance path planning problem as finding a sequence of states $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ from which the targets can be photographed. We assumed a minimum time state-to-state trajectory planner was available as a "black box" that could be accessed by our algorithms in order to evaluate the distance function $d(\mathbf{x}, \mathbf{x}')$, which is all we need to evaluate the goodness of any candidate solution. In this way, the minimum time state-to-state trajectory planner is a *module* within our algorithms. We could therefore use our algorithms with any of the minimum time state-to-state trajectory planners available in the literature. These include planners which can handle wind, no-fly zones, and any vehicle dynamics. The literature on nonholonomic trajectory planning is vast, so we survey only briefly a few works most relevant.

Without obstacles or wind, the procedure for computing an optimal Dubins pose-to-pose path was first shown using complicated measure theoretic arguments

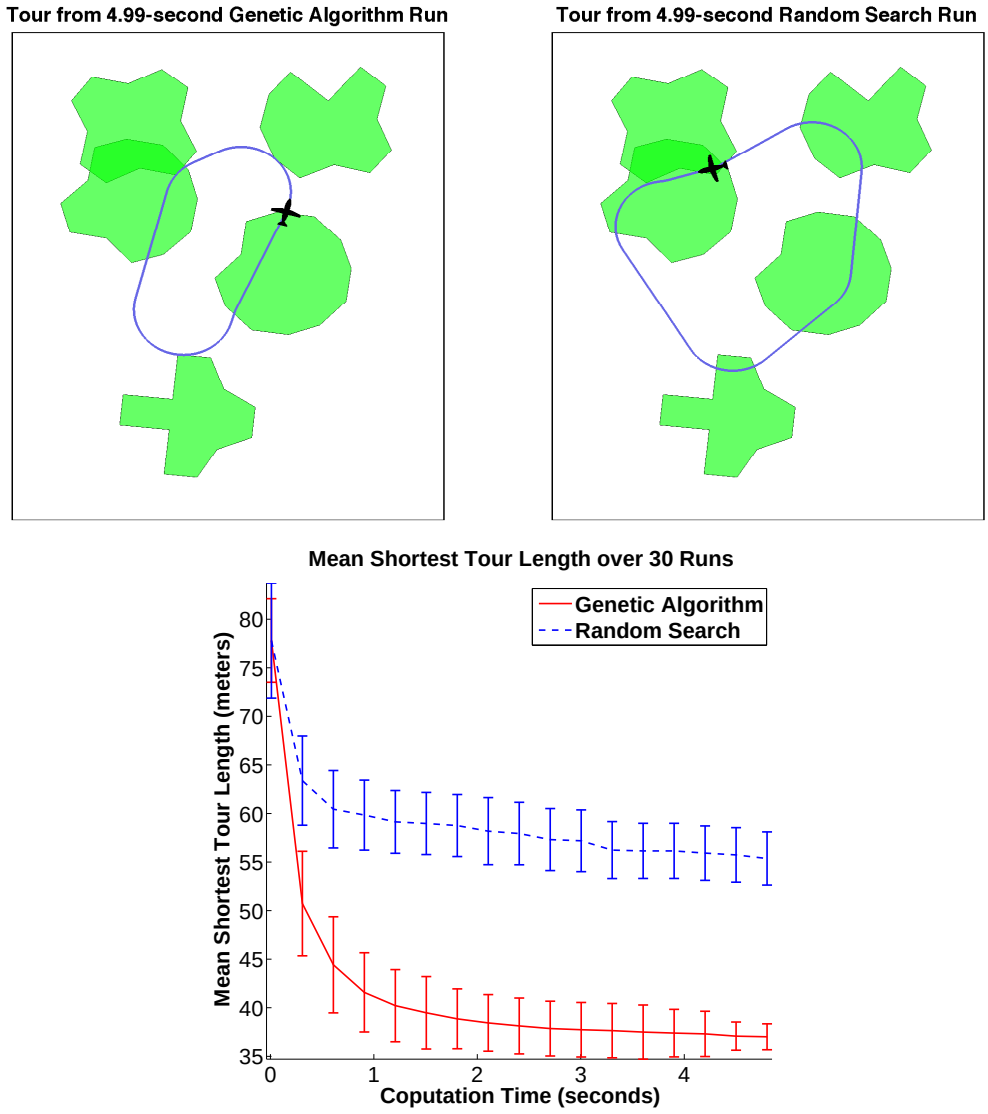



Figure 5.12. Computed example with $n = 5$ targets, aircraft minimum turn radius $r_{\min} = 3$ m. Green polygons represent the target visibility regions. Black dots are the tour nodes. Error bars show the sample standard deviations. Cf Table 5.6.

in [45], then later more concisely using Pontryagin's maximum principle from optimal control in [97]. More recently it has been shown that in a constant wind field without obstacles, a shortest pose-to-pose Dubins path can be caclulated in

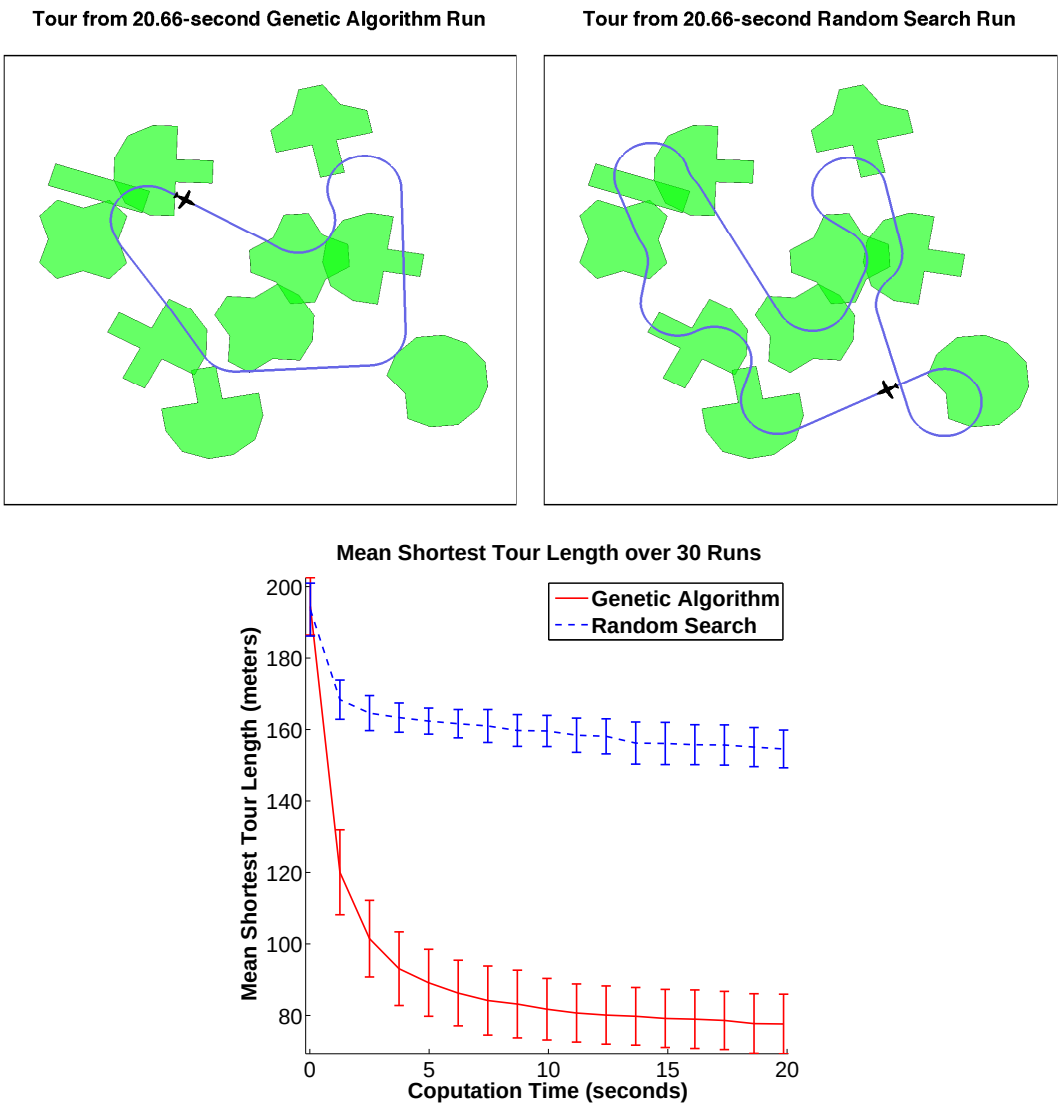



Figure 5.13. Computed example with $n = 10$ targets, aircraft minimum turn radius $r_{\min} = 3$ m. Green polygons represent target visibility regions. Black dots are the tour nodes. Error bars show the sample standard deviations. Cf Table 5.6.

constant time to fixed accuracy [46, 106, 107]. Using these methods, pose-to-pose shortest path queries with no obstacles can be computed in constant time.

Given a polygonal environment with polygonal holes represented by a total

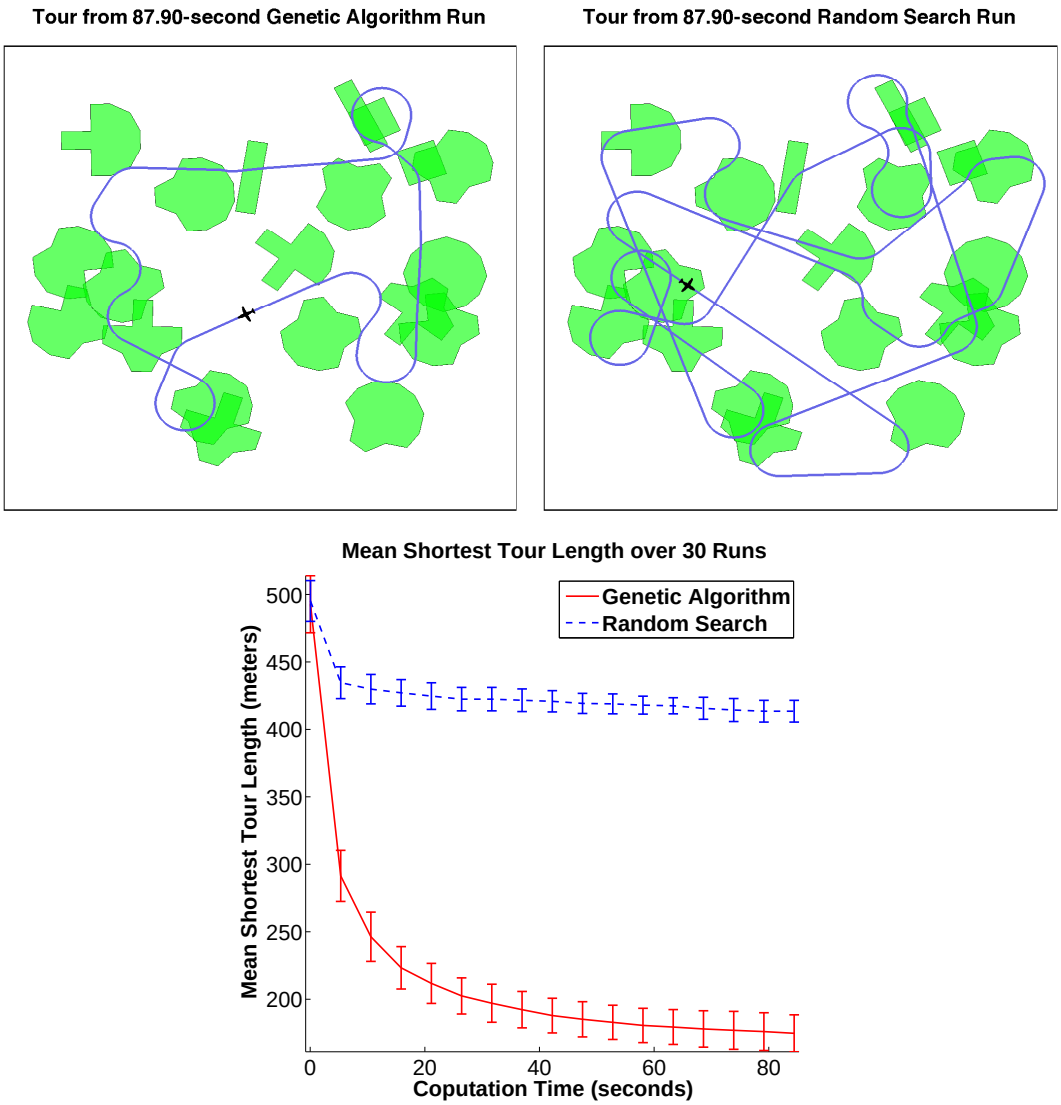



Figure 5.14. Computed example with $n = 20$ targets, aircraft minimum turn radius $r_{\min} = 3$ m. Green polygons represent target visibility regions. Black dots are the tour nodes. Error bars show the sample standard deviations. Cf Table 5.6.

of $m$ vertices, the shortest collision-free Euclidean path (no curvature constraint) can be calculated in $\mathcal{O}(m \log m)$ time [108]. Unfortunately the same problem with a Dubins vehicle is NP-hard in $m$ [109]. However, much work has been

done to quickly find nearly optimal obstacle avoiding paths, surveyed further in [110, 62, 18]. Trajectory planners specifically intended for fixed-wing UAVs, which use a branch and bound technique, are described in [111, 112]. Another approach to nonholonomic motion planning with obstacles is to use a MILP (Mixed Integer Linear Program) [113, 114].

### 5.5.2 Open-Path vs. Closed-Tour Problems

So far in this chapter, we have considered only closed-tour solutions to the reconnaissance UAV path planning problem. One may alternatively wish to find an open reconnaissance path from a fixed initial pose $\mathbf{x}_{\text{initial}}$ to a different fixed final pose $\mathbf{x}_{\text{final}}$. In this case, instead of the closed-tour cost function in Eq. 5.2, we use the *open path cost function*

$$C(\mathbf{x}_1, \ldots, \mathbf{x}_n) \;=\; d(\mathbf{x}_{\text{initial}}, \mathbf{x}_1) + \sum_{i=1}^{n-1} d(\mathbf{x}_i, \mathbf{x}_{i+1}) + d(\mathbf{x}_n, \mathbf{x}_{\text{final}}).$$

The genetic algorithm in Sec. 5.4 can directly be applied to open-path problems by simply replacing the closed-tour cost function with the open-path cost function. The sampling-based roadmap methods must be modified only slightly in how they construct the roadmap. The open-path roadmap has all the vertices and edges that the closed-tour roadmap of Def. 5.3 does, but in addition has two degenerate (single-vertex) clusters, one for the initial pose and one for the final pose. The initial degenerate cluster is connected by distance $d$-weighted edges outgoing to all vertices in nondegenerate clusters. The final degenerate cluster is connected (1) by a zero-weight edge outgoing to the initial cluster vertex, and (2) by distance $d$-weighted edges incoming from all vertices in the nondegenerate clusters.

## 5.6 Conclusion

We have formulated the general aircraft visual reconnaissance problem for static ground targets in terrain and shown that, under simplifying assumptions, it can be reduced to a variant of the Traveling Salesman Problem which we call the PVDTSP (Polygon-Visiting Dubins Traveling Salesman Problem). Although the PVDTSP reduces to the well-studied DTSP and ETSP in the *sparse limit* as targets are very far apart, our worst-case analysis demonstrated the importance of developing specialized algorithms for the PVDTSP in the *dense limit* as targets are close together. We designed two sampling-based roadmap methods for the PVDTSP. These methods operate by sampling finite discrete sets of vehicle states to approximate a PVDTSP instance by an FOTSP instance, then applying existing FOTSP algorithms. One of our sampling-based roadmap methods uses what is known as the Noon-Bean transformation and is *resolution complete*, which means it provably converges to a nonisolated global optimum as the number of samples grows. Our other sampling-based roadmap method achieves faster computation times by using an approximate dynamic programming technique, but consequently only converges to a nonisolated global optimum modulo target order. We also designed a genetic algorithm for the PVDTSP. The genetic algorithm has no performance guarantees but is easiest to implement and tends to find good feasible solutions quickly. Numerical experiments indicate that both the sampling-based roadmap methods and genetic algorithm deliver good solutions suitably quickly for online purposes when applied to PVDTSP instances having up to about 20 targets. Additionally, all algorithms allow trade-off of computation time for solution quality and are extensible to handle wind, airspace constraints, any vehicle dynamics, and open-path problems. The algorithms each have their

own merit depending on application and user needs. The methods could also be used in a receeding horizon fashion for stochastic scenarios with pop-up targets.

While the algorithms we have presented are essentially ready to be fielded, there is much room for future work. We are currently investigating extensions to multiple vehicles, constant factor approximation guarantees, a way to calculate how many samples a roadmap needs to guarantee a prescribed accuracy, and optimal ratios of roadmap orientation dispersion to position dispersion. Aside form improvements to the existing algorithms, it would be interesting to numerically evaluate hybrid approaches.

# Chapter 6

# Conclusion

In this dissertation we designed algorithms to solve visibility problems in motion coordination, planning, and control. Solutions were enabled by a unique blend of mathematical tools from the research domains of combinatorics, computational geometry, robot motion planning, and control theory. In Chapter 2 we presented the first distributed deployment algorithm which solves, with provable performance, the Distributed Visibility-Based Deployment Problem with Connectivity in polygonal environments with holes. The deployment algorithm was designed as a distributed emulation of a centralized incremental partition algorithm. Given at least $\lfloor \frac{n+2h-1}{2} \rfloor$ agents in an environment with $n$ vertices and $h$ holes, the deployment is guaranteed to achieve full visibility coverage of the environment in time $\mathcal{O}(n^2 + nh)$, or time $\mathcal{O}(n + h)$ if the maximum perimeter length of any vertex-limited visibility polygon in $\mathcal{E}$ is uniformly bounded as $n \to \infty$. The deployment behaved in simulations as predicted by the theory and can be extended to achieve robustness to agent arrival, agent failure, packet loss, removal of an environment edge (such as an opening door), or deployment from multiple roots.

In Chapter 3 we showed that the Searchlight Scheduling Problem can be re-

duced to a path planning problem through an appropriate information graph. The proof was based on an exact cell decomposition of the searchlights' toroidal configuration space. Using the reduction result we designed a complete algorithm for searchlight scheduling. The algorithm is divided into two parts. First, geometric preprocessing is performed in time polynomial in the number of guards and environment vertices. Second, the information graph is searched breadth-first. Our time complexity upper bound for the information graph breadth-first search is exponential in the output size. Although it remains an important open question whether the general Searchlight Scheduling Problem is NP-hard, computed examples demonstrated that the algorithm can be practical for problem instances of useful size, and for which there currently exists no other algorithm. Additionally, we have shown that our complete algorithm for searchlight scheduling can be directly extended to the $\phi$-Searchlight Scheduling Problem in which sensors have finite fields of view.

In Chapter 4 we provided two solutions to the distributed searchlight scheduling problem in polygonal environments without holes. In an environment with $n$ vertices and $r$ reflex vertices, DOWSS requires that the guards satisfy the standing assumptions, has message size $\mathcal{O}(n)$, and sometimes requires time $\mathcal{O}(r^2)$ to clear an environment. PTSS requires that the agents be positioned according to a PTSS tree, has constant message size, and requires time linear in the height of the PTSS tree. We have given two procedures for constructing PTSS trees, one requiring no more than $r \leq n-3$ guards for a general polygonal environment, and two requiring no more than $\frac{n-2}{2}$ guards for an orthogonal environment. Guards slew through a total angle no greater than $2\pi$, so the upper bounds on the time for PTSS to clear an environment with these partitions are $\frac{2\pi}{\omega}r \leq \frac{2\pi}{\omega}(n-3)$ and

$\frac{\pi}{\omega}(n-2)$, respectively. Because PTSS allows searchlights to slew concurrently, it generally clears an environment much faster than DOWSS. However, a direct comparison is not appropriate since DOWSS does not specify how to choose guards whereas PTSS does. To extend DOWSS and PTSS for environments with holes, one simple solution is to add one guard per hole, where a simply connected environment is simulated by the extra guards using their beams to connect the holes to the outer boundary. Another straightforward extension for PTSS would be to combine it directly with a distributed deployment algorithm such as in Chapter 2, so that deployment and searchlight slewing happen concurrently.

In Chapter 5 we formulated the general aircraft visual reconnaissance problem for static ground targets in terrain and showed that, under simplifying assumptions, it can be reduced to a variant of the Traveling Salesman Probem which we call the PVDTSP (Polygon-Visiting Dubins Traveling Salesman Problem). Although the PVDTSP reduces to the well-studied DTSP and ETSP in the *sparse limit* as targets are very far apart, our worst-case analysis demonstrated the importance of developing specialized algorithms for the PVDTSP in the *dense limit* as targets are close together. We designed two sampling-based roadmap methods for the PVDTSP. These methods operate by sampling finite discrete sets of vehicle states to approximate a PVDTSP instance by an FOTSP instance, then applying existing FOTSP algorithms. One of our sampling-based roadmap methods uses what is known as the Noon-Bean transformation and is *resolution complete*, which means it provably converges to a nonisolated global optimum as the number of samples grows. Our other sampling-based roadmap method achieves faster computation times by using an approximate dynamic programming technique, but consequently only converges to a nonisolated global optimum modulo target

order. We also designed a genetic algorithm for the PVDTSP. The genetic algorithm has no performance guarantees but is easiest to implement and tends to find good feasible solutions quickly. Numerical experiments indicate that both the sampling-based roadmap methods and genetic algorithm deliver good solutions suitably quickly for online purposes when applied to PVDTSP instances having up to about 20 targets. Additionally, all algorithms allow trade-off of computation time for solution quality and are extensible to handle wind, airspace constraints, any vehicle dynamics, and open-path problems. The algorithms each have their own merit depending on application and user needs. The methods could also be used in a receeding horizon fashion for stochastic scenarios with pop-up targets.

## 6.1 Future Directions

### Visibility Coverage

While our distributed deployment algorithm presented in Chapter 2 represents a significant theoretical advancement in nonconvex coverage, a practical implementation with actual robots would still present significant challenges, particularly to overcome our assumptions that (1) agents communicate, process, and establish a local common reference frame while moving, and (2) agents do not obstruct visibility or movement of other agents. Other interesting possibilities for future work in the area of deployment and nonconvex coverage include 3D environments, dynamic environments with moving obstacles, and optimizing different performance measures, e.g., based on continuous instead of binary visibility, or with minimum redundancy requirements.

## Visibility-Based Pursuit-Evasion

We hope that in the future either NP-hardness of the Searchlight Scheduling Problem can be shown, or else that the computational time complexity bounds for a complete algorithm can be tightened. There are also many interesting and unexplored variations of the Searchlight Scheduling Problem including minimizing time to clear the environment, evaders with bounded speed, sensor limitations such as limited depth of field, sensors sweeping a half-plane, or sweeping cones through 3D environments. Aside from searchlight scheduling, a particularly interesting problem in visibility-based purusit evasion, to our knowledge unsolved, is that of minimizing the time to perform a coordinated search given a limited number of mobile guards.

## Reconnaissance Path Planning for a UAV

The UAV path planning algorithms presented in Chapter 5 are essentially ready to be fielded, yet there is much room for future work. We are currently investigating extensions to multiple vehicles, constant factor approximation guarantees, a way to calculate how many samples a roadmap needs to guarantee a prescribed accuracy, and optimal ratios of roadmap orientation dispersion to position dispersion. Aside form improvements to the existing algorithms, it would be interesting to numerically evaluate hybrid approaches.

# Bibliography

[1] D. T. Lee and A. K. Lin, "Computational complexity of art gallery problems," *IEEE Transactions on Information Theory*, vol. 32, no. 2, pp. 276–282, 1986.

[2] S. Eidenbenz, C. Stamm, and P. Widmayer, "Inapproximability results for guarding polygons and terrains," *Algorithmica*, vol. 31, no. 1, pp. 79–113, 2001.

[3] M. R. Garey and D. S. Johnson, *Computers and Intractability.* Springer, 1979.

[4] A. Efrat and S. Har-Peled, "Guarding galleries and terrains," *Information Processing Letters*, vol. 100, no. 6, pp. 238–245, 2006.

[5] B. C. Liaw, N. F. Huang, and R. C. T. Lee, "The minimum cooperative guards problem on $k$-spiral polygons," in *Canadian Conference on Computational Geometry*, (Waterloo, Canada), pp. 97–102, 1993.

[6] J. Urrutia, "Art gallery and illumination problems," in *Handbook of Computational Geometry* (J. R. Sack and J. Urrutia, eds.), pp. 973–1027, North-Holland, 2000.

[7] J. O'Rourke, *Art Gallery Theorems and Algorithms*. Oxford University Press, 1987.

[8] T. C. Shermer, "Recent results in art galleries," *Proceedings of the IEEE*, vol. 80, no. 9, pp. 1384–1399, 1992.

[9] V. Chvátal, "A combinatorial theorem in plane geometry," *Journal of Combinatorial Theory. Series B*, vol. 18, pp. 39–41, 1975.

[10] S. Fisk, "A short proof of Chvátal's watchman theorem," *Journal of Combinatorial Theory. Series B*, vol. 24, p. 374, 1978.

[11] I. Bjorling-Sachs and D. Souvaine, "An efficient algorithm for guard placement in polygons with holes," *Discrete and Computational Geometry*, vol. 13, no. 1, pp. 77–109, 1995.

[12] F. Hoffmann, M. Kaufmann, and K. Kriegel, "The art gallery theorem for polygons with holes," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, (San Juan, Puerto Rico), pp. 39–48, Oct. 1991.

[13] G. Hernández-Peñalver, "Controlling guards," in *Canadian Conference on Computational Geometry*, (Saskatoon, Canada), pp. 387–392, 1994.

[14] V. Pinciu, "A coloring algorithm for finding connected guards in art galleries," in *Discrete Mathematical and Theoretical Computer Science*, vol. 2731/2003 of *Lecture Notes in Computer Science*, pp. 257–264, Springer, 2003.

[15] H. González-Baños and J.-C. Latombe, "A randomized art-gallery algorithm for sensor placement," in *ACM Symposium on Computational Geometry*, (Medford, MA), pp. 232–240, 2001.

[16] U. M. Erdem and S. Sclaroff, "Automated camera layout to satisfy task-specific and floor plan-specific coverage requirements," *Computer Vision and Image Understanding*, vol. 103, no. 3, pp. 156–169, 2006.

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

[18] 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, 2005.

[19] R. Simmons, D. Apfelbaum, D. Fox, R. Goldman, K. Haigh, D. Musliner, M. Pelican, and S. Thrun, "Coordinated deployment of multiple heterogenous robots," in *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, (Takamatsu, Japan), pp. 2254–2260, 2000.

[20] A. Howard, M. J. Matarić, and G. S. Sukhatme, "An incremental self-deployment algorithm for mobile sensor networks," *Autonomous Robots*, vol. 13, no. 2, pp. 113–126, 2002.

[21] S. Suri, E. Vicari, and P. Widmayer, "Simple robots with minimal sensing: From local visibility to global geometry," *International Journal of Robotics Research*, vol. 27, no. 9, pp. 1055–1067, 2008.

[22] A. Ganguli, J. Cortés, and F. Bullo, "Distributed deployment of asynchronous guards in art galleries," in *American Control Conference*, (Minneapolis, MN), pp. 1416–1421, June 2006.

[23] A. Ganguli, J. Cortés, and F. Bullo, "Visibility-based multi-agent deployment in orthogonal environments," in *American Control Conference*, (New York), pp. 3426–3431, July 2007.

[24] A. Ganguli, *Motion Coordination for Mobile Robotic Networks with Visibility Sensors*. PhD thesis, Electrical and Computer Engineering Department, University of Illinois at Urbana-Champaign, Apr. 2007.

[25] K. Sugihara, I. Suzuki, and M. Yamashita, "The searchlight scheduling problem," *SIAM Journal on Computing*, vol. 19, no. 6, pp. 1024–1040, 1990.

[26] M. Yamashita, I. Suzuki, and T. Kameda, "Searching a polygonal region by a group of stationary $k$-searchers," *Information Processing Letters*, vol. 92, no. 1, pp. 1–8, 2004.

[27] B. P. Gerkey, S. Thrun, and G. Gordon, "Visibility-based pursuit-evasion with limited field of view," *International Journal of Robotics Research*, vol. 25, no. 4, pp. 299–315, 2006.

[28] M. Yamashita, H. Umemoto, I. Suzuki, and T. Kameda, "Searching for mobile intruders in a polygonal region by a group of mobile searchers," *Algorithmica*, vol. 31, no. 2, pp. 208–236, 2001.

[29] B. Simov, G. Slutzki, and S. M. LaValle, "Pursuit-evasion using beam detection," in *IEEE Int. Conf. on Robotics and Automation*, (San Francisco, CA), pp. 1657–1662, Apr. 2000.

[30] J. H. Lee, S. M. Park, and K. Y. Chwa, "Simple algorithms for searching a polygon with flashlights," *Information Processing Letters*, vol. 81, no. 5, pp. 265–270, 2002.

[31] I. Suzuki and M. Yamashita, "Searching for a mobile intruder in a polygonal region," *SIAM Journal on Computing*, vol. 21, no. 5, pp. 863–888, 1992.

[32] L. J. Guibas, J. C. Latombe, S. M. LaValle, D. Lin, and R. Motwani, "A visibility-based pursuit-evasion problem," *International Journal of Computational Geometry & Applications*, vol. 9, no. 4-5, pp. 471–493, 1999.

[33] S. M. LaValle, B. H. Simov, and G. Slutzki, "An algorithm for searching a polygonal region with a flashlight," in *Proceedings of the 16th Annual Symposium on Computational Geometry*, pp. 260–269, 2000.

[34] I. Suzuki, Y. Tazoe, M. Yamashita, and T. Kameda, "Searching a polygonal region from the boundary," *International Journal of Computational Geometry & Applications*, vol. 11, no. 5, pp. 529–553, 2001.

[35] S. M. LaValle and J. E. Hinrichsen, "Visibility-based pursuit-evasion: the case of curved environments," *IEEE Transactions on Robotics and Automation*, vol. 17, no. 2, pp. 196–202, 2001.

[36] L. Guilamo, B. Tovar, and S. M. LaValle, "Pursuit-evasion in an unkown environment using gap navigation trees," in *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, (Sendai, Japan), pp. 3456–3462, Sept. 2004.

[37] T. Kameda, M. Yamashita, and I. Suzuki, "On-line polygon search by a seven-state boundary 1-searcher," *IEEE Transactions on Robotics*, vol. 22, no. 3, pp. 446–460, 2006.

[38] L. E. Parker, "Distributed algorithms for multi-robot observation of multiple moving targets," *Autonomous Robots*, vol. 12, no. 3, pp. 231–55, 2002.

[39] V. Isler, S. Kannan, and S. Khanna, "Randomized pursuit-evasion in a polygonal environment," *IEEE Transactions on Robotics*, vol. 5, no. 21, pp. 875–884, 2005.

[40] J. W. Durham, A. Franchi, and F. Bullo, "Distributed pursuit-evasion with limited-visibility sensors via frontier-based exploration," in *IEEE Int. Conf. on Robotics and Automation*, (Anchorage, Alaska), May 2010. To appear.

[41] B. Simov, G. Slutzki, and S. M. LaValle, "Clearing a polygon with two 1-searchers," *International Journal of Computational Geometry & Applications*, 2007. to appear.

[42] A. Efrat, L. J. Guibas, D. C. Lin, J. S. B. Mitchell, and T. M. Murali, "Sweeping simple polygons with a chain of guards," in *ACM-SIAM Symposium on Discrete Algorithms*, (San Francisco, CA), pp. 927–936, Jan. 2000.

[43] P. Chandler, M. Pachter, and S. Rasmussen, "Uav cooperative control," in *American Control Conference*, (Arlington, VA), pp. 50–55, June 2001.

[44] A. Ryan, M. Zennaro, A. Howell, R. Sengupta, and J. K. Hedrick, "An overview of emerging results in cooperative uav control," in *IEEE Conf. on Decision and Control*, 2004.

[45] L. E. Dubins, "On curves of minimal length with a constraint on average curvature and with prescribed initial and terminal positions and tangents," *American Journal of Mathematics*, vol. 79, pp. 497–516, 1957.

[46] T. G. McGee, S. Spry, and J. K. Hedrick, "Optimal path planning in a constant wind with a bounded turning rate," in *AIAA Conf. on Guidance, Navigation and Control*, (San Francisco, CA), Aug. 2005. Electronic Proceedings.

[47] J. T. Betts, "Survey of numerical methods for trajectory optimization,"

*AIAA Journal of Guidance, Control, and Dynamics*, vol. 21, no. 2, pp. 193–207, 1998.

[48] C. H. Papadimitriou, "The euclidean traveling salesman problem is np-complete," *Theoretical Computer Science*, vol. 4, pp. 237–244, 1977.

[49] J. L. Ny, E. Frazzoli, and E. Feron, "The curvature-constrained traveling salesman problem for high point densities," in *IEEE Conf. on Decision and Control*, pp. 5985–5990, 2007.

[50] K. E. Nygard, P. R. Chandler, and M. Pachter, "Dynamic network flow optimization models for air vehicle resource allocation," 2001.

[51] C. Schumacher, P. R. Chandler, and S. R. Rasmussen, "Task allocation for wide area search munitions," in *American Control Conference*, 2002.

[52] Z. Tang and Ü. Özgüner, "Motion planning for multi-target surveillance with mobile sensor agents," *IEEE Transactions on Robotics*, vol. 21, no. 5, pp. 898–908, 2005.

[53] S. Rathinam, R. Sengupta, and S. Darbha, "A resource allocation algorithm for multi-vehicle systems with non holonomic constraints," *IEEE Transactions on Automation Sciences and Engineering*, vol. 4, no. 1, pp. 98–104, 2007.

[54] K. Savla, E. Frazzoli, and F. Bullo, "Traveling Salesperson Problems for the Dubins vehicle," *IEEE Transactions on Automatic Control*, vol. 53, no. 6, pp. 1378–1391, 2008.

[55] J. Le Ny and E. Feron, "An approximation algorithm for the curvature-constrained traveling salesman problem," in *Allerton Conf. on Communications, Control and Computing*, (Monticello, IL), Sept. 2005.

[56] M. de Berg, J. Gudmundsson, M. J. Katz, C. Levcopoulos, M. H. Overmars, and A. F. van der Stappen, "TSP with neighborhoods of varying size," pp. 21–35, 2002.

[57] A. Dumitresco and J. S. B. Mitchell, "Approximation algorithms for TSP with neighborhoods in the plane," *Journal of Algorithms*, vol. 48, no. 1, pp. 135–159, 2003.

[58] C. S. Mata and J. S. B. Mitchell, "Approximation algorithms for geometric tour and network design problems," in *Symposium on Computational Geometry*, pp. 360–369, 1995.

[59] G. Gutin and A. P. Punnen, *The Traveling Salesman Problem and Its Variations*. Springer, 2007.

[60] C. E. Noon and J. C. Bean, "An efficient transformation of the generalized traveling salesman problem," Tech. Rep. 91-26, Department of Industrial and Operations Engineering, University of Michigan, Ann Arbor, 1991.

[61] M. Fischetti, J. J. Salazar-González, and P. Toth, *The Traveling Salesman Problem and its Variations*, ch. 13. Kluwer, 2002.

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

[63] P. Oberlin, S. Rathinam, and S. Darbha, "A transformation for a heterogeneous, multiple depot, multiple traveling salesmen problem," in *American Control Conference*, pp. 1292–1297, 2009.

[64] Z. Michalewicz and D. B. Fogel, *How to Solve It: Modern Heuristics*. Springer, 2004.

[65] B. Freisleben and P. Merz, "A genetic local search algorithm for solving symmetric and asymmetric traveling salesman problems," in *In Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*, pp. 616–621, IEEE Press, 1996.

[66] P. Larrañaga, C. M. H. Kuijpers, R. H. Murga, I. Inza, and S. Dizdarevic, "Genetic algorithms for the travelling salesman problem: A review of representations and operators," *Artificial Intelligence Review*, vol. 13, pp. 129–170, April 1999.

[67] L. V. Snyder and M. S. Daskin, "A random-key genetic algorithm for the generalized traveling salesman problem," *European Journal of Operational research*, vol. 174, 2006.

[68] Z. C. Huang, X. L. Hu, and S. D. Chen, "Dynamic traveling salesman problem based on evolutionary computation," in *Proceedings of the 2001 Congress on Evolutionary Computation*, vol. 2, pp. 1283–1288, 2001.

[69] W. Pullan, "Adapting the genetic algorithm to the travelling salesman problem," in *Proceedings of the 2003 Congress on Evolutionary Computation*, vol. 2, pp. 1029–1035, 2003.

[70] V. Shaferman and T. Shima, "Cooperative uav tracking under urban occlusions and airspace limitations," in *AIAA Conf. on Guidance, Navigation and Control*, (Honolulu, Hawaii), Aug 2008. Electronic Proceedings.

[71] V. Shaferman and T. Shima, "Co-evolution genetic algorithm for UAV distributed tracking in urban environments," in *ASME Conference on Engineering Systems Design and Analysis*, Jul 2008.

[72] H. H. Gonzalez-Banos, C.-Y. Lee, and J.-C. Latombe, "Real-time combinatorial tracking of a target moving unpredictably among obstacles," in *Proceedings IEEE International Conference on Robotics & Automation*, 2002.

[73] R. Murrieta-Cid, A. Sarmiento, S. Bhattacharya, and S. Hutchinson, "Maintaining visibility of a moving target at a fixed distance: The case of observer bounded speed," in *Proceedings IEEE International Conference on Robotics & Automation*, pp. 479–484, 2004.

[74] S. K. Ghosh, *Visibility Algorithms in the Plane*. Cambridge University Press, 2007.

[75] E. Trucco and A. Verri, *Introductory Techniques for 3-D Computer Vision*. Prentice Hall, 1998.

[76] D. A. Forsyth and J. Ponce, *Computer Vision: A Modern Approach*. Prentice Hall, 2002.

[77] J. Bittner and P. Wonka, "Visibility in computer graphics," *Environment and Planning B: Planning and Design*, vol. 30, pp. 729–756, Sept. 2003.

[78] F. Bullo, J. Cortés, and S. Martínez, *Distributed Control of Robotic Networks*. Applied Mathematics Series, Princeton University Press, 2009. Available at http://www.coordinationbook.info.

[79] D. Cruz, J. McClintock, B. Perteet, O. A. A. Orqueda, Y. Cao, and R. Fierro, "Decentralized cooperative control: A multivehicle platform for research in networked embedded systems," *IEEE Control Systems Magazine*, vol. 27, no. 3, pp. 58–78, 2007.

[80] N. J. Nilsson, "A mobile automaton: An application of artificial intelligence techniques," in *1st International Conference on Artificial Intelligence*, pp. 509–520, 1969.

[81] K. J. Obermeyer, "The VisiLibity library." `http://www.VisiLibity.org`, 2008. R-1.

[82] B. J. Moore and K. M. Passino, "Distributed task assignment for mobile agents," *IEEE Transactions on Automatic Control*, vol. 52, no. 4, pp. 749–753, 2007.

[83] M. M. Zavlanos, L. Spesivtsev, and G. J. Pappas, "A distributed auction algorithm for the assignment problem," in *IEEE Conf. on Decision and Control*, pp. 1212–1217, Dec. 2008.

[84] K. J. Obermeyer, A. Ganguli, and F. Bullo, "A complete algorithm for searchlight scheduling," *International Journal of Computational Geometry & Applications*, Oct. 2008. Submitted.

[85] K. J. Obermeyer, A. Ganguli, and F. Bullo, "Asynchronous distributed

searchlight scheduling," in *IEEE Conf. on Decision and Control*, (New Orleans, LA), pp. 4863–4868, Dec. 2007.

[86] H. Choset, E. Acar, A. A. Rizzi, and J. Luntz, "Exact cellular decompositions in terms of critical points of Morse functions," in *IEEE Int. Conf. on Robotics and Automation*, (San Francisco, CA), pp. 2270–2277, Apr. 2000.

[87] H. Choset, "Nonsmooth analysis, convex analysis, and their applications to motion planning," *International Journal of Computational Geometry & Applications*, vol. 9, no. 4-5, pp. 447–469, 1999.

[88] J. O'Rourke, *Computational Geometry in C*. Cambridge University Press, 2000.

[89] D. Halperin, "Arrangements," in *Handbook of Discrete and Computational Geometry* (J. E. Goodman and J. O'Rourke, eds.), pp. 529–562, New York: Chapman and Hall/CRC Press, 2 ed., 2004.

[90] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2 ed., 2003.

[91] L. Guibas, D. Salesin, and J. Stolfi, "Epsilon geometry: building robust algorithms from imprecise computations," in *ACM Symposium on Computational Geometry*, (New York), pp. 208–217, June 1989.

[92] M. Segal, "Using tolerances to guarantee valid polyhedral modeling results," in *Proceedings of SIGGRAPH*, (Dallas, TX), pp. 105–114, Aug. 1990.

[93] K. J. Obermeyer, "Path planning for a UAV performing reconnaissance of static ground targets in terrain," in *AIAA Conf. on Guidance, Navigation and Control*, (Chicago, IL), Aug. 2009.

[94] K. J. Obermeyer, P. Oberlin, and S. Darbha, "Sampling-based roadmap methods for a visual reconnaissance UAV," in *AIAA Conf. on Guidance, Navigation and Control*, (Toronto, ON, Canada), Aug. 2010. To Appear.

[95] K. J. Obermeyer, P. Oberlin, and S. Darbha, "Sampling-based roadmap methods for a visual reconnaissance UAV," *AIAA Journal of Guidance, Control, and Dynamics*, 2010. Submitted.

[96] J. Gross and J. Yellen, *Handbook of Graph Theory*. CRC Press, 2003.

[97] J.-D. Boissonnat, A. Cérézo, and J. Leblond, "Shortest paths of bounded curvature in the plane," *Journal of Intelligent and Robotic Systems*, vol. 11, pp. 5–20, 1994.

[98] J. E. Bresenham, "Algorithm for computer control of a digital plotter," in *Seminal Graphics: Pioneering Efforts that Shaped the Field*, (New York, NY, USA), pp. 1–6, Association for Computing Machinery, 1998.

[99] H. Niederreiter, *Random Number Generation and Quasi-Monte Carlo Methods*. No. 63 in CBMS-NSF Regional Conference Series in Applied Mathematics, Society for Industrial & Applied Mathematics, 1992.

[100] J. H. Halton, "On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals," *Numerische Mathematik*, vol. 2, pp. 84–90, 1960.

[101] D. Applegate, R. Bixby, V. Chvátal, and W. Cook, "On the solution of traveling salesman problems," in *Documenta Mathematica, Journal der Deutschen Mathematiker-Vereinigung*, (Berlin, Germany), pp. 645–656,

Aug. 1998. Proceedings of the International Congress of Mathematicians, Extra Volume ICM III.

[102] K. Helsgaun, "An effective implementation of the linkernighan traveling salesman heuristic," *European Journal of Operational Research*, vol. 126, pp. 106–130, October 2000.

[103] D. L. Applegate, R. E. Bixby, and V. Chvátal, *The Traveling Salesman Problem: A Computational Study*. Applied Mathematics Series, Princeton University Press, 2006.

[104] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2 ed., 2001.

[105] J. C. Spall, *Introduction to Stochastic Search and Optimization*. Wiley-Interscience, 2003.

[106] R. McNeely, R. V. Iyer, and P. Chandler, "Tour planning for an unmanned air vehicle under wind conditions," *AIAA Journal of Guidance, Control, and Dynamics*, vol. 30, no. 5, pp. 1299–1306, 2007.

[107] L. Techy and C. A. Woolsey, "Minimum-time path planning for unmanned aerial vehicles in steady uniform winds," *AIAA Journal of Guidance, Control, and Dynamics*, vol. 32, no. 6, pp. 1736–1746, 2009.

[108] J. Hershberger and S. Suri, "An optimal algorithm for euclidean shortest paths in the plane," *SIAM Journal on Computing*, vol. 28, pp. 2215–2256, 1999.

[109] J. Reif and H. Wang, "The complexity of the two dimensional curvature-constrained shortest-path problem," in *In Proc. Third International Workshop on the Algorithmic Foundations of Robotics*, pp. 49–57, 1998.

[110] J.-C. Latombe, *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[111] A. Eele and A. Richards, "Path-planning with avoidance using nonlinear branch-and-bound optimisation," in *AIAA Conf. on Guidance, Navigation and Control*, (Hilton Head, South Carolina), 2007. Electronic Proceedings.

[112] A. Eele and A. Richards, "Comparison of branching strategies for path-planning with avoidance using nonlinear branch-and-bound," in *AIAA Conf. on Guidance, Navigation and Control*, (Honolulu, Hawaii), 2008. Electronic Proceedings.

[113] F. Borrelli, D. Subramanian, A. U. Raghunathan, and L. T. Biegler, "MILP and NLP techniques for centralized trajectory planning of multiple unmanned air vehicles," in *American Control Conference*, 2006.

[114] A. Richards and J. P. How, "Aircraft trajectory planning with collision avoidance using mixed integer linear programming," in *American Control Conference*, pp. 1936–1941, 2002.

# Appendix A

# Distributed Searchlight Scheduling Detailed Pseudocodes

More detailed versions of the pseudocodes in Tables 4.1 and 4.2.

**Name:** DOWSS

The root initially has state = 4 and all other agents begin with state = 1, where possible states are $1, 2, 3, \ldots, 12$. Each agent $i \in \{1, \ldots, N\}$, possesses local variables parent, child, $\psi_{\text{parent}}$, $\psi_{\text{temp}}$, $j$, $G$, $\Phi$, $\phi_{\text{start}}$, $\phi_{\text{finish}}$, $\Psi$, and $u$, all initially empty.

**SPEAK**

**while** state == 7 or state == 8 **do**
  {request help}
  BROADCAST($i, \psi_j(t)$, `help`);
  state $\leftarrow$ 8;
**while** state == 2 **do**
  {volunteer to help}
  BROADCAST($i$, parent$(t)$, $\psi_{\text{temp}}(t)$, `volunteer`);
  state $\leftarrow$ 3;
**while** state == 9 **do**
  {engage a child}
  BROADCAST($i$, child$(t)$, $\psi_j(t)$, `selected`);
  state $\leftarrow$ 10;
**while** state = 12 **do**
  {report to parent when complete}
  BROADCAST($i$, parent, $\psi_{\text{parent}}(t)$, `complete`);
  state $\leftarrow$ 1;

**LISTEN**

**while** state == 1 or state == 3 **do**
  {listen for help request}
  **if** RECEIVE($i', \psi_{j'}^{[i']}(t-\tau)$, `help`), where $0 \le \tau \le \delta$ **then**
    $\psi_{\text{temp}} \leftarrow \psi_{j'}^{[i']}(t-\tau)$; state $\leftarrow$ 3;
**while** state == 8 **do**
  {listen for volunteers}
  **if** RECEIVE($i'$, parent$^{[i']}(t-\tau), \psi_{\text{temp}}^{[i']}(t-\tau)$, `volunteer`), where $0 \le \tau \le \delta$, parent$^{[i']}(t-\tau) = i$, and $\psi_{\text{temp}}^{[i']}(t-\tau) = \psi_j$ **then**
    child $\leftarrow i'$; state $\leftarrow$ 9;
**while** state == 3 **do**
  {listen for engagement by parent}
  **if** RECEIVE($i'$, child$^{[i']}, \psi_{j'}^{[i']}(t-\tau)$, `selected`), where $0 \le \tau \le \delta$, where child$^{i'} = i$ **then**
    parent $\leftarrow i'$; $\psi_{\text{parent}} \leftarrow \psi_{j'}^{[i']}(t-\tau)$; state $\leftarrow$ 4;
  **else if** RECEIVE($i'$, child$^{[i']}$, $\psi_{j'}^{[i']}(t-\tau)$, `selected`), where $0 \le \tau \le \delta$, where child$^{[i']} \neq i$ **then**
    state $\leftarrow$ 1;
**while** state == 10 **do**
  {listen for child to report completion}
  **if** RECEIVE($i'$, parent$^{[i']}, \psi_{\text{parent}}^{[i']}(t-\tau)$, `complete`), where $0 \le \tau \le \delta$ **then**
    **if** $j < m$ **then**
      $j \leftarrow j+1$; state $\leftarrow$ 6;
    **else if** $j == m$ **then**
      state $\leftarrow$ 11;

**PROCESS**

**while** state == 3 **do**
  {use $\psi_{\text{temp}}$ and $\mathcal{V}(p^{[i]})$ to check if able to help}
  **if** able to see across oriented polyline $\psi_{\text{temp}}$ into semiconvex subregion and not located in interior of that subregion **then**
    state $\leftarrow$ 2;
**while** state == 4 **do**
  {when first engaged, perform geometric computations; note visibility gaps are listed ccw and radially outwards}
  Compute $\phi_{\text{start}}$ and $\phi_{\text{finish}}$; {start and finish angles}
  Compute $G \leftarrow (g_1, \ldots, g_m)$; {visibility gaps}
  Compute $\Phi \leftarrow (\phi_1, \ldots, \phi_m)$; {resp. angles of visibility gaps}
  Compute $\Psi \leftarrow (\psi_1, \ldots, \psi_m)$; {polyline for each visibility gap}
  $j \leftarrow 1$; {initialize rotation counter}
  state $\leftarrow$ 5;

**ROTATE**

**while** state == 5 **do**
  {aim at start angle and switch searchlight on}
  $\theta^{[i]} \leftarrow \phi_{\text{start}}$;
  state $\leftarrow$ 6;
**while** state == 6 **do**
  {rotate to next angle}
  **while** $\theta^{[i]} < \phi_j$ **do**
    $u \leftarrow \frac{\min\{u_{\max}, ||\phi_j - \theta^{[i]}||\}}{||\phi_j - \theta^{[i]}||}(\phi_j - \theta^{[i]})$;
    $\theta^{[i]} \leftarrow \theta^{[i]} + u$;
  state $\leftarrow$ 7;
**while** state == 11 **do**
  {rotate to finish angle and switch searchlight off}
  **while** $\theta^{[i]} < \phi_{\text{finish}}$ **do**
    $u \leftarrow \frac{\min\{u_{\max}, ||\phi_{\text{finish}} - \theta^{[i]}||\}}{||\phi_{\text{finish}} - \theta^{[i]}||}(\phi_{\text{finish}} - \theta^{[i]})$;
    $\theta^{[i]} \leftarrow \theta^{[i]} + u$;
  state $\leftarrow$ 12;

**Name:** PTSS

The root initially has state = 2 and all other agents begin with state = 1, where possible states are $1, 2, \ldots, 10$. Each agent $i \in \{0, \ldots, N-1\}$ possesses local variables parent, $\Phi$, $\phi_{\text{start}}$, $\phi_{\text{finish}}$, $C$, $j$, and $u$, all initially empty. As needed to clarify ownership, a superscript with square brackets indicates the UID of the agent to whom a variable belongs.

**SPEAK**

1: **while** state == 7 **do**
2: {tell child to aim across gap}
3: BROADCAST($\text{child}_j$, aim_across_gap);
4: state $\leftarrow$ 8;
5: **while** state == 4 **do**
6: {tell parent when aimed across gap}
7: BROADCAST($i$, aimed_across_gap);
8: state $\leftarrow$ 5;
9: **while** state == 9 **do**
10: {tell child when finished rotating over gap}
11: BROADCAST($\text{child}_j$, gap_passed);
12: **if** $j < m$ **then**
13: $j \leftarrow j + 1$; state $\leftarrow$ 6;
14: **else if** $j == m$ **then**
15: state $\leftarrow$ 10;

**LISTEN**

1: **while** state == 1 **do**
2: {listen for instruction from parent to aim across gap}
3: **if** RECEIVE($\text{child}^{[i']}$, aim_across_gap) and $i == \text{child}^{[i']}$ **then**
4: state $\leftarrow$ 2;
5: **while** state == 8 **do**
6: {listen for confirmation from child aimed across gap}
7: **if** RECEIVE($i'$, aimed_across_gap) and $i' == \text{child}_j$ **then**
8: $j \leftarrow j + 1$; state $\leftarrow$ 6;
9: **while** state == 5 **do**
10: {listen for confirmation that parent has passed the gap}
11: **if** RECEIVE($\text{child}^{[i']}$, gap_passed) and $i == \text{child}^{[i']}$ **then**
12: state $\leftarrow$ 6;

**PROCESS**

1: **while** state == 2 **do**
2: {when first engaged, perform geometric computations}
3: Compute $\phi_{\text{start}}$ and $\phi_{\text{finish}}$; {start and finish angles}
4: Compute $\Phi \leftarrow (\phi_1, \ldots, \phi_m)$; {ordered gap endpoint angles}
5: Compute $C \leftarrow (\text{child}_1, \ldots, \text{child}_m)$; {resp. child UIDs}
6: $j \leftarrow 1$; {initialize rotation counter}
7: state $\leftarrow$ 3;

**ROTATE**

1: **while** state == 3 **do**
2: {aim at start angle and switch searchlight on}
3: $\theta^{[i]} \leftarrow \phi_{\text{start}}$;
4: state $\leftarrow$ 4;
5: **while** state == 6 **do**
6: {rotate to next angle}
7: **while** $\theta^{[i]} < \phi_j$ **do**
8: $u \leftarrow \frac{\min\{u_{\max}, ||\phi_j - \theta^{[i]}||\}}{||\phi_j - \theta^{[i]}||}(\phi_j - \theta^{[i]})$;
9: $\theta^{[i]} \leftarrow \theta^{[i]} + u$;
10: **if** $j$ is odd **then**
11: state $\leftarrow$ 7;
12: **else if** $j$ is even **then**
13: state $\leftarrow$ 9
14: **while** state == 10 **do**
15: {rotate to finish angle and switch searchlight off}
16: **while** $\theta^{[i]} < \phi_{\text{finish}}$ **do**
17: $u \leftarrow \frac{\min\{u_{\max}, ||\phi_{\text{finish}} - \theta^{[i]}||\}}{||\phi_{\text{finish}} - \theta^{[i]}||}(\phi_{\text{finish}} - \theta^{[i]})$;
18: $\theta^{[i]} \leftarrow \theta^{[i]} + u$;
19: state $\leftarrow$ 1;