Visibility-Aware Motion Planning

by

Gustavo Nunes Goretkin

M. Eng. EECS, MIT (2016)
S.B. EECS and Mathematics, MIT (2013)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

February 2022

© Gustavo Nunes Goretkin 2022. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Author . . . .
Department of Electrical Engineering and Computer Science
January 14, 2022

Certified by . . . .
Leslie Pack Kaelbling
Professor of Computer Science and Engineering
Thesis Supervisor

Certified by . . . .
Tomás Lozano-Pérez
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Visibility-Aware Motion Planning

by Gustavo Nunes Goretkin

Submitted to the Department of Electrical Engineering and Computer Science on January 14, 2022, in partial fulfillment of the requirements for the degree of Doctor of Philosophy

ABSTRACT

The motion planning problem, of deciding how to move to achieve a goal, is ubiquitous in robotics. In many robotics applications, there is a map of the environment that is generally useful, but typically outdated as it does not include information about unknown obstacles, such as clutter. This thesis addresses the problem of planning for a robot with an onboard obstacle-detection sensor. The planning objective is to remain safe with respect to unknown obstacles by guaranteeing that the robot will not move into any region of the workspace before observing it.

Although much work has addressed a version of this problem in which the field of view of the sensor is a sphere around the robot, we address robots with a limited field of view, which may arise from sensor limitations or self-occlusions in the case of mobile manipulation robots. We provide a formal definition of the problem, which we call Visibility-Aware Motion Planning (VAMP), and several solution methods with different computational trade-offs. We demonstrate the behavior of these planning algorithms in illustrative planar domains. The key to an efficient solution is to aggressively prune paths, while ensuring that the overall search strategy is sound and complete.

We demonstrate that motion planning problems like VAMP benefit from a path-dependent formulation, in which the state at a search node is represented implicitly by the path to that node. The straightforward approach to computing the feasibility of a successor node in such a path-dependent formulation takes time linear in the path length to the node, in contrast to a (possibly very large) constant time for a more typical search formulation. For long-horizon plans, this linear-time computation for each node becomes prohibitive. To improve upon this, we introduce the use of a fully persistent spatial data structure (FPSDS). We apply a FPSDS to VAMP search, by using a nearest-neighbor data structure to perform bounding-volume queries. We demonstrate an asymptotic and practical improvement in the runtime of finding VAMP solutions in large domains. To the best of our knowledge, this is the first use of a fully persistent data structure for accelerating motion planning.

Thesis Supervisor: Tomás Lozano-Pérez
Title: Professor of Computer Science and Engineering

Thesis Supervisor: Leslie Pack Kaelbling
Title: Professor of Computer Science and Engineering

Acknowledgements

It is overwhelming to consider all the individuals whose journeys have meaningfully intersected mine during the course of this project, this degree, this graduate school experience, this most recent 40% of my life, which I spent at MIT and in Cambridge, and this lifetime. I've left this difficult task to the very end, and I'll just have to aspire to do a decent job, because there are truly too many people to thank.

I would like to first and foremost thank my advisors Leslie and Tomás. In February 2010, I joined the Learning and Intelligent Systems group as an undergraduate researcher, and now it's finally time to say goodbye. As if that wasn't already a long time ago, Leslie was my freshman-year (September 2009!) academic advisor, and I met Tomás that first semester as well. Their impressive commitment to doing research and the time they dedicate to meeting with students is nothing short of incredible. I am lucky to have had their mentorship. They have given me liberty to stumble and fall, and have stepped in when I needed help getting back up. I am especially appreciative for their gentle recommendation to work on the subject of this dissertation, a problem they encountered while conducting their own research. I will miss their patience and encouragement and willingness to help when I was struggling.

Thanks to our group's administrative assistant Teresa Cataldo for her cheerfulness and thorough work, and to Professor Nick Roy for serving on the committee.

When I first joined the lab, I worked most closely with George Konidaris. He is the single person who is most responsible for my having decided to pursue graduate school. I am grateful for his kind and helpful mentorship during those early formative years.

Thank you to my fellow lab mates of LIS past, present, and future. I enjoyed the morning walks, Post-It note shenanigans (eye'm watching you), dogs, escape rooms, surviving a virtual zombie apocalypse, karaoke, video games, side projects, technical discussions, commiseration, and more. Thank you for the memories, Alejandro Perez, Alex LaGrassa, Ariel Anders, Beomjoon Kim, Brian Axelrod, Caelan Garrett, Caris Moses, Clément Gehring, Dylan Hadfield-Menell, Ferran Alet, Gilwoo Lee, Jenny Barry, Lawson L.S. Wong, Patrick Barragán, Rachel Holladay, Rodrigo Gomes, Rohan Chitnis, Sam Davies, Tom Silver, Yoonchang Sung, Zelda Marriet, Zhutian Yang, Zi Wang

I'd like to thank the undergraduate students with whom I've had the opportunity to work, Eric Chen, and Hanxiang Ren, Silvia Knappe. Sathwik Karnik deserves special mention for tolerating me for three years, ultimately leading to the data structure applications present in this dissertation.

I had the great privilege of being part of starting the American Sign Language and Deaf Culture club while I was a graduate student. Thank you to Barbara Johnson and Kristina T. Johnson for their efforts in organizing ASL classes and events. Nilma Dominque reminded me to cultivate my heritage, and enriching my cultural experience on campus with Brazilian music and food, and Portuguese language. Obrigado!

There were many benchmarks I was concerned with in grad school. I wanted to write faster, clearer, shorter code, and publish many papers, and do internships, and be well-rounded, and have hobbies, and so on. It has felt and continues to feel impossible at times to handle all these hopes, disappointments, dreams, fears, existential crises, etc. I am humbled to have friends who believed in me when I did not, and who kept me company especially in these recent months and years that have been particularly

challenging for personal and global reasons. To Ari, Caris, Clément, Diyang, Dorothy, Emily Wean, Grace, Kamran, Makeita, Melissa, Nirmala (All these years later, I finally drank the castor oil.), Stephan, Tharu & Rishi, and Emma and Xiaolu and others, thank you.

I cannot thank my family enough. For everything. To my parents Eleonora and Guilherme, for their bold decision to immigrate when I was two years old, forever changing my life, for all the care and love, and especially for amplifying my curiosity. To my brother Gui for being a role model, and teaching me so much, and with such kindness. To my sister-in-law Laura for withholding her full celebration until all aspects of the submission process of this document were completed. To my family here, there, and everywhere, eu amo vocês.

Finally, no one has been as patient with me as Oscar. I don't have the words to describe the support he has given me. I am so lucky to have met you, dear.

Contents

Contents 5

1 Introduction **11**
1.1 The VAMP problem 11
1.2 Contributions 14
Problem setting 14
Related work 15
Search approach 15
Unifying related path search problems 16
Formulation 16
Algorithms 17
Data structures 18
Summary 18

2 Background **19**
2.1 Planning approach to robot behavior 19
2.2 Related work 21
Overview 21
Motion Planning 22
Coverage problems 22
Exploration problems 24
Navigation problems 24
Planning for perception 26
2.3 Assumptions 26
Ball of visibility 26
Safety criteria 28
Observation model 28
Motion model 28
Environment model 29
Search and execution strategies 29

3 Path Search **31**
3.1 Introduction 31
3.2 Pruning without affecting correctness 33
3.3 Generalizing pruning 33
3.4 Shortest paths 33
3.5 Self-avoiding walks 34
3.6 Informative walks 34
3.7 Vertex visit graphs 35
3.8 Complexity of vertex visit graphs 37
Quadratic scaling of solution length 37

Polynomial-time feasibility checking for reversible graphs . . . 37
NP-completeness of vertex visit graphs . . . 37
3.9 Configuration path-dependent search . . . 38
Algorithm . . . 39
Configuration path-dependent problems . . . 39

4 Formulation and Algorithms **43**
4.1 Formulation . . . 43
Formulation of subproblem . . . 46
4.2 Algorithms for visibility-aware motion planning . . . 46
Forward heuristic search in belief space . . . 47
Reverse search . . . 49
Searches with aggressive pruning . . . 50
Tree-visibility tour . . . 51
Visibility preimage backchaining . . . 53
Heuristic for exploration . . . 54
4.3 Reformulation . . . 59
Pruning . . . 59
Constraint relaxation . . . 60
Goal regression . . . 61
4.4 Complexity . . . 62
Complexity of problem class . . . 62
Complexity of solution length . . . 62
Complexity of algorithm . . . 63
4.5 Experiments and planning results . . . 64
4.6 Appendix: Geometry . . . 65
Representing the sensing region . . . 65
Visibility computations . . . 66
Visibility constraint versus collision detection . . . 66
Representing convex polyhedra . . . 67
Visible region by constructing shadow volumes . . . 68
Constructing unseen swept region . . . 68
Representing the visible region of a path . . . 69
Discrete workspace . . . 70
Post-processing to minimize views . . . 72
4.7 Appendix: Extension to tactile sensing . . . 73
Direct reduction to VAMP domains . . . 74
Extension of safety constraint . . . 74
Visual and tactile sensing . . . 75
Temporal constraints and objectives . . . 75

5 Data Structures for Efficient Planning **76**
5.1 Introduction . . . 76
5.2 Traditional applications of spatial data structures . . . 76
5.3 Motion Planning Applications . . . 78
Minimum Constraint Removal . . . 78

Belief-Space Planning 79
Visibility-Aware Motion Planning 80
5.4 Fully Persistent Nearest-Neighbor Tree 80
Insertion 81
Range Query 82
Complexity Analysis 83
Comparison with Baseline 83
5.5 VAMP Problem Formulation 84
5.6 Relaxed VAMP Solution 85
5.7 Efficient Visibility Queries 86
Bounding Volumes 86
5.8 Experiments 87
Experimental Results 88
5.9 Discussion 89

6 Discussion **91**
6.1 Execution 91
6.2 Issues arising in replanning 91
Commitment 91
6.3 Replanning 92

List of Figures

1.1 Some possible sensed volumes with respect to robot configuration. 11
1.2 Two views of the same robot configuration. The shaded region cannot be seen by the head-mounted sensor in this configuration. 11
1.3 In the HallwayEasy domain, a robot with narrow (30°) field of view must steer carefully around a corner. . . . 13
1.4 In the HallwayHard domain, a robot with a wide field of view, must enter the hallway camera-first, back out, re-orient, and back in. 14
1.5 The TwoHallway domain with hand-generated configuration subgoals, labeled 1–4. 14
1.6 An example minimum vertex visit graph 16
1.7 Comparison of a fully persistent and ephemeral data structures. Circles represent states of the data structures, and blue shapes represent data inserted into the data structure. Traditionally, data structures are *ephemeral*, meaning that updates are destructive. In a fully persistent data structure, an update to the data structure generates a new version, and also maintains the previous version. The previous version is available both for querying and for alternate updates. 18
2.1 A fundamental diagram that establishes the notion of robot and not-robot (world, plant, environment). . . . 19
2.2 Robots that plan use some model, almost always with less fidelity than reality, of themselves and the world. . 19
2.3 A common paradigm for designing robot behavior. This architecture leaves room for the goal to be specified, possibly by a higher-level system. 20
2.4 An environment in which there is no solution for a single *pur*suer to catch an evader. 23
3.1 Two distinct walks ("N"and "S" shapes) with the same start and end configurations, visiting the same configurations along the way. 34
3.2 Each of the 6 informative paths on the $(2, 2)$ lattice induces a graph with edges labeled with subsets of vertices. . . 36
3.3 The same vertex visit graph can be induced by two informative paths. 36
3.4 The same vertex visit graph can be induced by two informative paths of the same length. 37

3.5 An example polynomial-time reduction from 3-SAT to finding a shortest path on a vertex visit graph. 38

4.1 Illustration of violation region X. 46
4.2 Level sets and heatmap of field F used to compute heuristic for acquiring visibility of the white points in the region surrounding $(1, 2)$. 56
4.3 Behavior of visibility heuristic 56
4.4 Two levels of relaxed planning with a visibility goal. . 57
4.5 Difficult examples for the Vamp_Backchain algorithm. . 58
4.6 Domain illustrating worst-case (quadratic) scaling of solution length with domain size. 63
4.7 Domain illustrating back-and-forth solution path in visibility-based pursuit evasion. 64
4.8 Set difference for union of convex polyhedra represented by half-spaces. 69
4.9 Some discretized swept volumes for a planar T-shaped robot. 72
4.10 For cell t to be visible from the center of cell s, the cell below t must be clear. 72
4.11 (a) Some possible sensed volumes with respect to robot configuration (blue) and environment (gray). The orange region can be covered with an unretractable cane. A few configurations of a cane are shown in black. The yellow region can be covered with a depth sensor (or retractable cane). (b) A robot with vision (yellow) and bump (red) sensing in a domain with start at s and goal at g. 74

5.1 Example vamp problem instance. In the domains we discuss, the viewcone (in yellow) is fixed relative to the robot. 80
5.2 Example of part of a fully persistent tree with information stored at each node. The red text at each node denotes the changes from its parent node. The subsequence of pointers – s_1, s_2, s_3, and s_4 – in the persistent tree nodes is shown in the order of allocations in the memory pool. Here, $M = 5$. 81
5.3 Motivating example of a VAMP problem instance in which computing the unseen swept region may be expensive. The violation-free path requires the robot to look through the glass wall into the hallway containing q_{goal}. The path found from the relaxed vamp problem results in unseen swept regions. 85

5.4 Trajectory $[q_1, q_2, q_3, q_4]$ with swept region $S(q_3, q_4)$ from q_3 to q_4. The viewcones $V(q)$ are shown with bounding radius r_{vis}. For $V(q_1)$, we show that the bounding ball has center $w_{v,1}$. The swept region $S(q_3, q_4)$ is shown with a bounding ball $\mathcal{B}(w_{s,4}, r_{s,4})$. 86
5.5 Experiment domains: (a) OneHallway, (b) HorseshoeHallway, and (c) GlassHallway. 87
5.6 These plots show the results for the OneHallway and HorseshoeHallway domains. The left column shows the overall runtimes and total times spent in Find_Vis_Viol. The right column shows the memory storage (via `Base.summarysize`) of the search trees. Discontinuities in memory use as the length grows are attributed to Julia data structure implementation details [90]. 89
5.7 These plots show similar statistics as shown in Figure 5.6 but for the GlassHallway domain. 89
6.1 In this environment, there are two walls that are detected to have length a and b. The walls continue beyond the figure, possibly indefinitely. If both walls are infinitely long, then there is no solution path from the initial to goal configuration. 91
6.2 Illustration of replanning in an unknown environment. 94

List of Tables

2.1 Related motion planning problems. 23
3.1 The counts of informative paths for a few modest lattice sizes. 35
4.1 Planning results across varying viewcones 65
5.1 Comparison of amortized time complexities. 83

1 Introduction

1.1 The VAMP problem

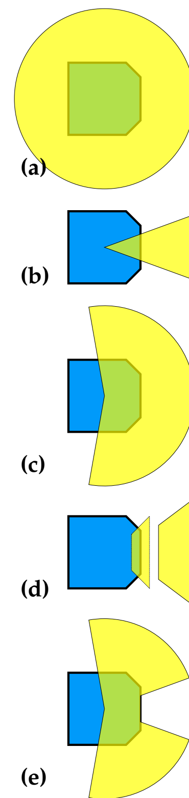


Figure 1.1: Some possible sensed volumes with respect to robot configuration.

Consider a robot designer tasked with developing a mobile-manipulation robot system that moves within an environment that contains obstacles. Supposing a set of assumptions, which we postpone describing for the sake of introduction, then a robot designer faces an algorithmic problem: given some input data describing the robot and environment, generate a path for the robot to follow. If the input data includes an accurate-enough representation of the obstacles in the environment, then the algorithmic problem is a *traditional motion planning* problem. In addition to *known* obstacles, the environment generally contains *unknown* obstacles — obstacles that are not represented in the input data — and in this setting, the robot must in general incorporate sensing into its plan in order to guarantee that it will not collide with any obstacles while it follows the planned path to the goal.

In one extreme instance of this problem, the environment is entirely unknown (and unchanging), and the best strategy might involve a separate phase of exploring and building a map. If this cartography exercise succeeds, then all obstacles are known obstacles, and the setting becomes amenable to traditional motion planning.

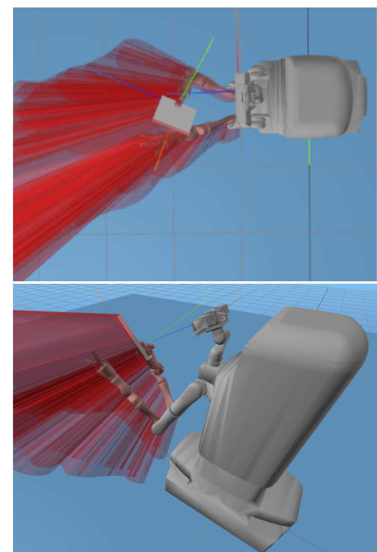


Figure 1.2: Two views of the same robot configuration. The shaded region cannot be seen by the head-mounted sensor in this configuration.

We will focus on a different regime that arises in, e.g., the case of a household robot. Many obstacles in the domain (e.g. walls, countertops) are known, but there are other unknown temporary obstacles (e.g., toys, trash cans, chairs). In this case, it is worthwhile to plan a path to a target *configuration*, optimistically. This path must avoid known obstacles, as is the case in traditional motion planning. In addition, the path must take the robot's sensing capability into account. By executing the planned path, any region of work space that the robot moves into will have been (optimistically observed earlier in the path. Said differently, the robot never moves into a region of space that it has not already observed and therefore verified to be empty. Should the robot encounter an unknown (unexpected) obstacle during the course of executing the plan, it can then make a new plan that takes into account this new obstacle. This new plan may incorporate additional sensing, avoid the obstacle, or move it out of the way. We call this strategy "optimistic" because the generated plans do not contain contingencies for the unknown obstacles. Nonetheless, the strategy is safe because it ensures that

the robot system will notice the need for a contingency should one arise.

When we speak of sensing, we mean any robot-mounted ability to gather information about the locations of obstacles in its neighborhood. The sensor could be based on camera imaging, lidar, or even the ability to reach out slowly with a hand and detect contact or lack thereof (see 4.7). The key limitation of these sensing modalities is occlusion; the sensor cannot report occupancy information if there is an obstacle in the line of sight. Therefore, we name this class of motion planning problems VAMP, for *visibility-aware motion planning*.

Traditional motion planning is a well-studied problem that is simpler than (i.e. subsumed by) the problem that we focus on in this thesis. It is nonetheless, in general, a challenging problem that continues to benefit from research. [1]

If a robot's sensors grant it visibility of a ball (a region in *work space*) that completely includes the robot in all configurations, and if the environment is static (unchanging), and the robot is quasi-static (the robot can immediately stop in place) then the problem of safe movement is simple. In such an instance of VAMP, any path can be executed safely. That's not to say that the robot system need not monitor unexpected obstacles, but the problem can be decoupled. No planning foresight is needed to ensure that paths acquire perspectives to guarantee safety with respect to unknown obstacles.

Figure 1.1 illustrates several sensor configurations for a non-articulated planar mobile robot. Case (a) reflects the most common assumption about sensing: that is, that the robot can perceive a ball in the work space, and this ball is a superset of the robot; case (b) shows a narrow view as might occur for a fixed vision sensors; case (c) shows a wide field of view as might occur with some steerable sensors; case (d) occurs for many humanoid robots with a camera mounted on the head: although they can see a view cone in front of them, it is occluded by the body and so there is a region of space immediately in front of the robot that cannot be seen; and case (e) illustrates a situation in which, for example, a humanoid is carrying a large box in front of it, so its field of view is split into two narrow cones.

Figure 1.2 demonstrates a mobile-manipulation robot with a head-mounted sensor; the robot is in a configuration in which its arms cause self-occlusions. A robot designer might be tempted to assert that the robot should just not navigate in such a precarious configuration; it should move its arms to the side, or perhaps it should put down the object that is responsible for blocking its view of the environment. However, the robot designer might reflect upon the experience of carrying a large box, perhaps while taking some stairs, to understand that there are important scenarios in which the

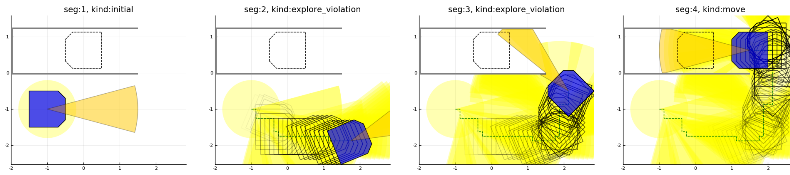


Figure 1.3: In the HallwayEasy domain, a robot with narrow (30°) field of view must steer carefully around a corner.

assertion is untenable. Misjudging the location of a stair step, or the presence of a slippery or painful object (e.g. a LEGO brick!) can have dire consequences. And so upon reflection, the designer remembers adopting a conservative strategy involving moving slowly, of course, but crucially also glancing past the box when and where possible.

There are many situations with less-than-perfect visibility. When the limitation is anticipated, good design might overcome it in practice. In a car, there are mirrors, headlights, windshield wipers, and also rearview cameras, proximity sensors, and increasingly sophisticated technology in the name of assisting or augmenting the driver's visibility. However, if, during a critical moment, the windshield fogs up, or if the wipers are ineffective against ice or a smudge, most drivers can mitigate the unfortunate situation by peering out the driver-side window. With any luck, a driver never experiences the need for such extreme improvisation, and driver's education and licensure does not directly prepare drivers in this way. In this situation where it was not possible to anticipate or rectify the limitations through design, it is the everyday experience of human embodiment that enables a driver to improvise. We desire general behavior from robots to cope in the face of limited visibility. Robot systems should have the ability to perform deliberate motions to acquire visibility.

For a large robot with a limited view that is navigating in a cluttered environment, the two problems, 1. moving to the goal and 2. observing to guarantee safety, are generally inextricably linked. Figure 1.3 shows a plan for a robot with a narrow view cone (shown in darker orange) to enter a hallway. The goal is depicted in the dashed outline. The yellow shading indicates the region of work space that has been seen by the plan. There is an initial visible region surrounding the region occupied by the robot at the initial configuration, without which there is no motion that would be safe. Note that the robot has to rotate back and forth as it moves, and swing wide around the corner, in order to guarantee safety. Figure 1.4 shows a robot with a wider field of view that must enter a narrow hallway backwards. Because it cannot turn around inside the hallway, it must first look down the hallway then back out and turn around. Some situations require fairly intricate solutions.

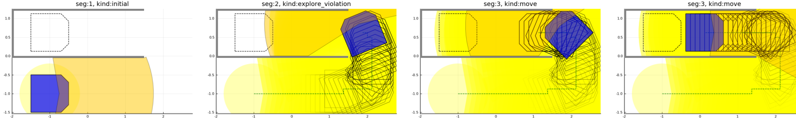


Figure 1.4: In the HallwayHard domain, a robot with a wide field of view, must enter the hallway camera-first, back out, re-orient, and back in.

Figure 1.5 shows a particularly difficult environment, in which the robot must enter one hallway to look through a narrow opening to gain visibility of a second hallway before entering that hallway. The figure denotes some key configurations. The goal configuration is marked 4. When the robot is at 3, it cannot continue to acquire full visibility of the goal configuration, due to e.g. a configuration-space obstacle that allows the robot to enter backwards only (which is possible to construct in a 3D work space). Alternatively, "fog" (an occluder but not a collider) may block. In either case, to make the goal configuration visible, the robot must be at 2 *and* 3. Finally, to make 2 visible, the robot must be at 1. The overall solution path for this domain may need to visit the same configuration multiple times, but these four subgoals partition the solution path into subpaths that do not revisit the same configuration. In Chapter 4 we discuss an algorithm that is incomplete (there are VAMP problems that they cannot solve, despite the existence of a solution). This algorithm can plan efficiently to visit these subgoals in order, if they are provided. This example illustrates that VAMP problems can have both short-range dependencies (which are relatively easy to handle), but also long-range dependencies.

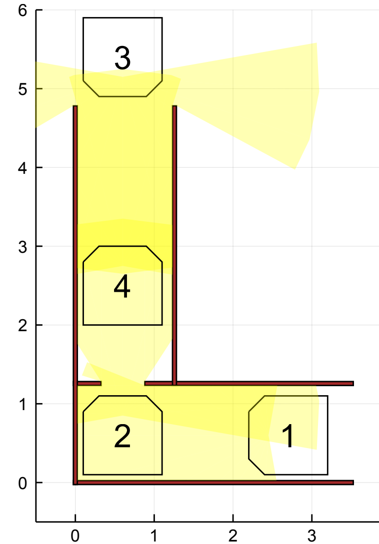


Figure 1.5: The TwoHallway domain with hand-generated configuration subgoals, labeled 1–4.

1.2 Contributions

Problem setting

Though they may in general be interleaved, we distinguish between "planning" and "execution". We assume that the robot knows a map in advance, e.g. a floor plan of a building. In other words, this map representation is available during the planning computation. This map is assumed to be accurate in the sense that the obstacles it contains are definitely present in the world. However, we wish to guarantee the robot's safety with respect to additional obstacles (furniture, trash, etc.) that may also be present in the world, but not in the map. We take for granted that it is not viable to model these additional obstacles ahead of time, and that information about these obstacles is first made available to the robot during the course of executing a plan. Even though we do not assume the environment during execution to match the environment during planning, we do assume in this work that the environment is static.

Related work

There are many problems in robotics that are related to ours, either due to similarity of solution approaches, or due to dealing with environment uncertainty and visibility. We address these relations, and clarify differences. For example, there is considerable work in structuring the planning computation so that computation can be re-used in the event of re-planning. The D* algorithm addresses replanning due to unknown obstacles [2], however this category of work addresses problem settings where there is a ball of visibility.

Search approach

We rely on what we call a *path-dependent* formulation. The problem of finding a shortest collision-free path does not benefit from such a formulation because we can represent a state as a configuration. In this case, the search procedure does not need to maintain multiple paths to a particular robot configuration. Instead, the search can keep only the path with the lowest cost. Alternatively, the search can keep only the first path found to a particular robot configuration. Path-dependent search problems require a search strategy that can maintain multiple paths to each reached configuration. One solution is to simply perform a search over all paths: the search state is the path, not the configuration. In this case, it is not necessary to keep backpointers to a node's predecessors — that information is available explicitly in the search state. In many problems, however, there is a short description that captures sufficient information about the path. We call this a *path summary*. In this case, it is only necessary to consider distinct path summaries to a given configuration, which greatly reduces the number of paths that must be considered. This notion of a path summary, with respect to a path, is analogous to the notion of "belief state", with respect to a sequence of actions and observations in the POMDP formulation. For VAMP, the path summary is exactly a region of work space that has been previously observed. Straightforward application of forward search to VAMP using full visibility as the path summary, even with very aggressive pruning and heuristics, is only feasible (in terms of computation) for impractically small domains as shown in Section 4.2. In practice, we must rely on backwards search, and sort of "counterfactual" solutions that relax constraints.

An example of a search algorithm that keeps only the lowest-cost path is Bellman-Ford.

For some search strategies, keeping only the first path to a configuration guarantees a shortest path to every configuration, e.g. Breadth-First Search, or A*with a consistent heuristic. These search strategies are correct and complete if there are no edges that can decrease the total cost (e.g. negative-weight edges)

A configuration is not necessarily a (search) state.

A "belief state" is almost always a probability distribution over some underlying states. A more general term is "information state", which we discuss in Section 2.2.

Unifying related path search problems

The state-space search formulation of planning unifies many problems that one may encounter in developing robot algorithms. Our formulation of VAMP as a problem, and our formulation of algorithms for solving VAMP problems makes use of standard concepts such as pruning search nodes based on a domination criterion, and pruning even more aggressively to search over only a subset of possible solutions. Though these concepts are standard, we take the opportunity to define them carefully in Chapter 3. We introduce an abstraction that represents some features of VAMP we call vertex-visit constrained graphs (see Figure 1.6). When appropriate, we describe concepts in this abstract and digestible setting.

In these terms, we explain several search problems such as Risk-Aware Motion Planning [3], Minimum-Constraint Removal [4], Minimum-Risk (Motion) Planning[5], Visibility-Based Pursuit Evasion [6], Minimum Swept-Volume Motion Planning, and Belief-Space Planning [7].

Formulation

The planners we present for VAMP are used in a "trust but verify" replanning framework, in which we assume, optimistically, for the purposes of planning, that the obstacles in our current map are, in fact, the only obstacles. This assumption makes it worthwhile to try to plan a complete path to the goal. However, because we are not certain that these are the only obstacles and because we wish to guarantee the robot's safety, we will seek a *visibility-aware* path to the goal, in which the robot never moves into space that has not been observed (and therefore verified) to be free during some previous part of its path. The robot could then execute this path until it observes an obstacle that invalidates the path. At that point, the execution program would insert that obstacle into its map and re-plan. The focus of this thesis is on methods for planning optimistic visibility-aware trajectories, though we illustrate execution in Chapter 6.

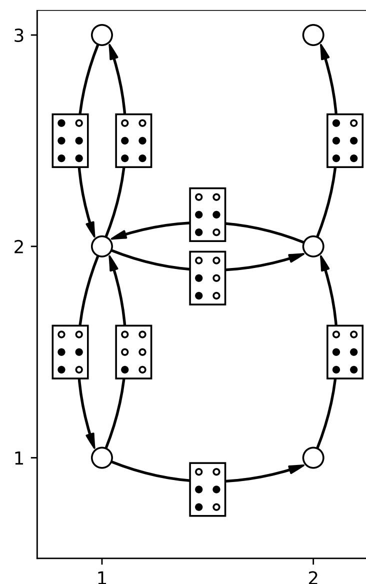


Figure 1.6: The minimum vertex visit graph corresponding to the path [(1, 1), (1, 2), (2, 2), (1, 2), (1, 1), (2, 1), (2, 2), (1, 2), (1, 3), (1, 2), (2, 2), (2, 3)]

Each arc is labeled with the set of vertices that must be visited before traversing the arc.

Our formulation allows a general visibility function that maps robot configurations to regions of work space that are observed. There are existing problem classes that subsume VAMP, e.g. POMDP, or belief-space planning. It is nonetheless beneficial to formulate VAMP as an important special case. We were initially surprised not to find a clear statement of this problem, let alone an adequate solution to this well-specified algorithmic problem. It is less surprising, perhaps, when

considering that the problem arises from the requirement of a robot capable of manipulation and navigation in some class of unforeseen circumstances. Some mobile robots have an all-encompassing ball of visibility in configuration space, and therefore do not need to do VAMP. Mobile manipulation robots do not, and they have not been deployed in many settings. Although the examples we presented are for a robot with 3D configuration space in a 2D work space, the formulation or algorithms are stated in general terms. This careful distinction between the configuration space and work space is a key feature of our approach. We assume that the robot has some form of obstacle sensor, but make no assumptions about it except that, for any configuration of the robot, it can observe some (possibly disjoint) subset of the work space and that this visibility function is known in advance, during the planning computation. We assume that observation and control are deterministic and that the robot always knows its configuration. In particular, we do not address the problem of localizing within a given map or handling stochasticity in sensor measurements.

A differential drive robot with a circular footprint and visibility of only the space directly in front of it does not need VAMP, either. Forward motion is safe, because of the sensing region, and rotational motion is safe because it does not change the workspace region occupied by the robot.

Algorithms

Solving the VAMP problem is quite computationally complex, because the state of the planning problem must somehow represent the region of work space that has been observed. Solutions to VAMP problems are generally paths that a traditional motion-planner would never generate, because the solution paths revisit the same robot configuration with different visibility states. The principal concern in developing an effective search algorithm is to reduce the size of the search tree via aggressive pruning. We develop several algorithms in relation to VAMP, which occupy different points in the trade-off space including path objectives, planning time, and completeness. In particular, we present in Section 4.2 an algorithm that is well-suited to solving VAMP problems. This algorithm is provably correct (will not generate an illegal plan); and is complete for robots that can reverse their motion without sweeping through additional work space. This assumption holds for holonomic robots, and also some non-holonomic robots, e.g. differential-drive or Ackermann-drive mobile robot bases, but it is not true for e.g. fixed-wing aircraft or a Dubins car that cannot go in reverse. Other algorithms we present serve to illustrate the belief-space nature of the VAMP problem, serve to efficiently determine reachability and feasibility of a problem instance, or serve as useful subprocedures for the algorithm in Section 4.2.

Data structures

Determining the feasibility (or priority) of candidate successors during planning time requires visibility queries, in addition to collision queries. These visibility queries can account for a substantial amount of the planning computation time, so we explore spatial data structures for accelerating these queries. Sample-based motion planning algorithms use spatial data structures to accelerate nearest-neighbor queries, but in that context, there is conceptually a single data structure that is shared by the planning computation. In our context, each search node in a planning tree must represent its visible region of work space, and conceptually each has its own spatial data structure. If generating a successor node involves copying, and then updating, this data structure, then these data structures could offer no asymptotic (and likely no practical) improvements for running time. Therefore, we require a substantially more intricate, so-called "fully persistent" data structure (see Figure 1.7 for a graphical summary) that maintains multiple versions of itself, one for each branch in the planning tree, while still maintaining some time and space benefits with respect to a brute-force solution. We take advantage of the unique capabilities of fully persistent to accelerate queries during planning in Chapter 5. Whereas in Chapter 4, a main focus is to reduce the number of paths present in a search tree, in this chapter the main focus is to reduce the computational effort for each path in the search tree.

In cases where collision objects are static during a planning episode, a spatial data structure can be constructed to accelerate collision queries. In this case, the data structure does not need to support any modifications during planning.

Summary

Chapter 2 discusses a wide body of related work. Chapter 3 introduces path-dependent search on a simple, abstract problem. Chapter 4 provides multiple solution strategies to our planning problem; it is the core of this dissertation. Chapter 5 provides a method for accelerating our planning algorithm with spatial data structures. Chapter 6 demonstrates the use of the planning algorithm in a re-planning loop as the environment is incrementally discovered.

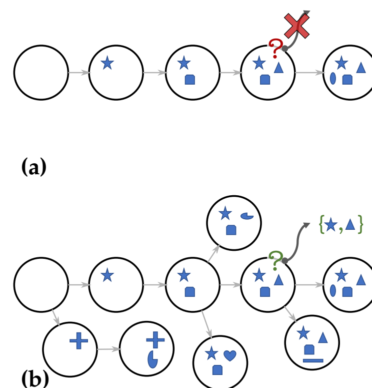


Figure 1.7: Comparison of a fully persistent and ephemeral data structures. Circles represent states of the data structures, and blue shapes represent data inserted into the data structure. Traditionally, data structures are *ephemeral*, meaning that updates are destructive. In a fully persistent data structure, an update to the data structure generates a new version, and also maintains the previous version. The previous version is available both for querying and for alternate updates.

2 Background

Fundamentally, a robot receives observations and takes actions. There are many ways to design a robotic system with such an interface. There are many approaches for how to select actions given observations and goals, and a robotic system may have several different approaches operating together. For some tasks, it is conceivable to hand-write a controller (e.g. Proportional-Integral-Derivative (PID), finite-state machine (FSM)). Such a solution is appropriate when it is possible to achieve goals with straightforward behaviors, but such a solution is unlikely to generalize to unanticipated situations. Alternatively, a controller could be synthesized from a declarative specification (e.g. Linear-Quadratic Regulator (LQR), Markov Decision Process (MDP)). Loosely speaking, in this case the designer chooses a parameterized family of controllers and uses a systematic way (e.g. minimizing an objective or satisfying a specification) of arriving at parameter values.

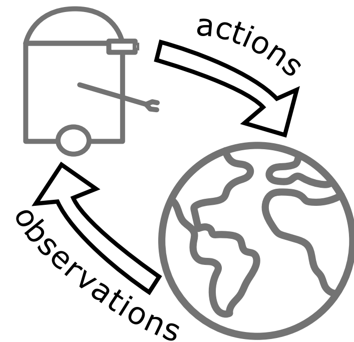


Figure 2.1: A fundamental diagram that establishes the notion of robot and not-robot (world, plant, environment).

In recent years, there is growing popularity in applying *reinforcement learning* as directly as possible to the observation/action interface. When there is no model available, this is essentially the only applicable approach.

We are interested in the more traditional approach, where there are multiple modules (some of which may be designed by learning). The interfaces between modules are chosen with considerations toward computational capabilities (e.g. summarize high bitrate camera images), human-interpretability (e.g. for debugging), and to encourage modularity (e.g. replace one manufacturer's robot arm with another's).

2.1 Planning approach to robot behavior

The work in this thesis does not directly determine what power to apply to robot actuators, nor how to interpret sensor signals. Those responsibilities belong to other subsystems.

A planning-based approach is natural given the requirements to generalize over the geometric data about the environment and the robot. This approach can lead to a general solution to a well-specified problem, even in tricky situations. Consider again Figure 1.5. It is

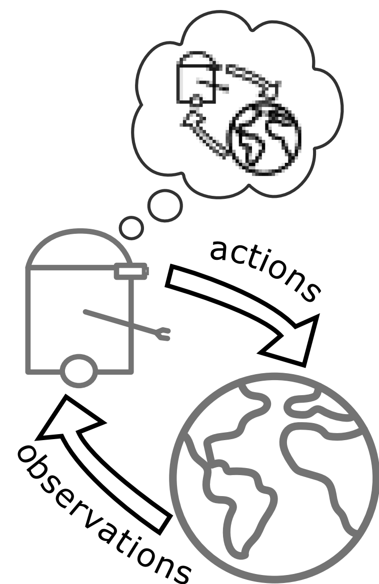


Figure 2.2: Robots that plan use some model, almost always with less fidelity than reality, of themselves and the world.

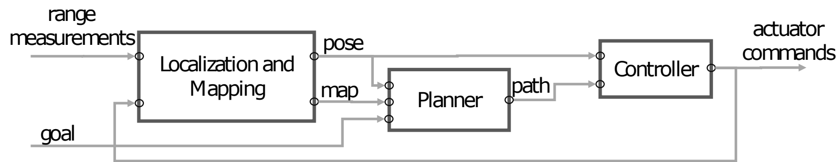


Figure 2.3: A common paradigm for designing robot behavior. This architecture leaves room for the goal to be specified, possibly by a higher-level system.

hard to imagine such a path arising from an approach unless it includes a component of planning.

In a robot system designed to operate in real-life general environments, geometric data about the environment and the robot's location within it comes from a perception and estimation subsystem that incorporates sensor data. Similarly, the path generated by the planner must be executed by a controller. Figure 2.3 demonstrates a common architecture for a robotic system.

It is important to distinguish between *planning* and *execution* phases. We must ultimately have a feedback policy to determine actions based on [a history of] observations. In some cases, such as for a deterministic, static world, a sequence of open-loop actions is sufficient to solve the task. In other cases, generating contingencies for all possible evolutions of the world is not feasible, and so we may make assumptions during planning time, and validate those assumptions during execution, replanning as necessary if they are violated. While we decompose robotic systems into these sense-plan-act modules, the concerns of the planner cannot be completely decoupled from the concerns of the adjacent systems. In general, the planner should be aware of limitations in perception and control. A plan that a robot cannot follow (for instance because it requires that a car-like vehicle move laterally, or because it requires a plan to fly in reverse) is of limited value. A plan that leads the robot to featureless regions where it loses track of its position in the world is similarly ill-advised. Section 2.2 provides examples of such work. Our VAMP problem is in the same spirit, to ensure that the mapping system acquires the relevant range measurements to produce a map that is valid.

Depending on the particular problem setting, there may be one or more sensible forms that the solution can take. In some cases (e.g., coverage, or navigation with a distribution over maps and a probabilistic guarantee) it is possible to find a single open-loop trajectory that solves the problem, e.g. [8]. In other cases, the trajectory must depend on the information that the robot gathers during execution. This may be handled by executing a policy online (which might be found by an off-line algorithm, such as a POMDP solver, or hand-constructed, as in the case of the Bug algorithm [9]) or by

interleaving planning and execution in the style of *receding-horizon control*, e.g. [10].

2.2 Related work

Overview

There are many avenues for connecting VAMP to other problems in robotics. Robot motion planning problems vary considerably in their objectives and assumptions about the problem. "Obstacle/collision avoidance", which is also the goal of motion planning, typically denotes the feedback policy setting rather than the planning setting. Our work focuses on uncertainty about the presence of obstacles, and there is other work to address other forms of uncertainty. Some of this work shares in common with VAMP a solution that that can be characterized as "optimism in the face of uncertainty", in which a complete path to the goal is planned optimistically. There are also computational geometry problems that are related by virtue of dealing with notions of visibility and coverage. Finally, there are several problems that share in common with VAMP a solution that involves *path-dependent* search on a graph of configurations. We summarize those problems in Section 3.9.

"Navigation" typically suggests motion planning for a mobile robot on a plane, but our setting applies to planning for general robots (and is in fact motivated by planning for mobile manipulation robots). VAMP is a *navigation* problem not an exploration problem; i.e. it finds a path to a goal configuration, not a path that, e.g., acquires coverage of some kind (some of our algorithms generate exploration subproblems to be solved, however). Additional key features of VAMP are that

- the solution is a *path*, not e.g. a set of views nor a controller,
- the collision sensing region depends on the robot configuration, and
- the legality of motions is governed by the sensing.

The last two points have a circular relation. Moving requires sensing, and sensing requires moving. The reason we perform motion planning is for collision avoidance. However, almost all the work involving collision avoidance in unknown environments, regardless of whether it addresses navigation or exploration problems, assumes a ball of visibility in configuration space that encompasses the robot. This assumption (described in more detail in Section 2.3)

is necessary to decoupling the circular relationship between motion and perception, and critically we do not make this assumption in the VAMP setting.

Motion Planning

Classical motion planning can be characterized by the assumption of perfect, complete, and metric model of quasi-static actuation, information about obstacles, the environment, and the goal. Under this setting, there is no need for sensing and feedback control.

This setting sometimes goes by the name *Generalized Mover's Problem* [1]. The solution to classical motion planning is a path in a robot's configuration space.

Generally planning is required not for a single rigid body (such as a piano), but for an articulated robot.

Kinodynamic motion planning relaxes the assumption that the robot can move quasi-statically through the configuration space. *Feedback motion planning* relaxes the assumption of perfect actuation by generating a path and also a state feedback controller. *Sensor-based motion planning* relaxes the assumption of perfect obstacle information. Examples include the so-called bug algorithms, which require only robot-local information [9, 11]. *Conformant planning* relaxes the assumption of perfect initial state information, using an open-loop plan (no feedback) that relies on funneling properties of the dynamics.

Under the assumption of perfect actuation and robot state information, we can organize the remaining motion planning problems into the following two orthogonal features: 1. the goal is a set in configuration space, or to observe a region of work space, and 2. the environment (for the sake of determining collisions) is known, or must be sensed. Table 2.1 provides a summary, with terminology.

Coverage problems

Coverage problems generally assume a known map of obstacles but require planning a path that will observe or touch all parts of the reachable space, for example, to inspect a structure, mow a lawn, mill a part, or vacuum a room [8, 12–15]

There are several problem statements in the purview of computational geometry that include a notion of visibility. The *Watchman Route Problem* [16], determine a shortest path for one "watchman" to cover a target region. *Visibility-based Pursuit Evasion* [6, 17] determines paths that achieve coverage in a stricter sense. When one

In the sense of coverage in *information space*, not work space [18, 19].

Table 2.1: Related motion planning problems. "c-space" denotes goals that are sets of configuration space. "i-space" stands for "information space" and denotes goals that are e.g. sets of workspace to be covered. An asterisk (*) denotes a wildcard. For example, the solution to a navigation problem could be either an open-loop path, or a feedback controller.

Problem Setting	solution	goal	obstacles	collision detection	localization
Classical motion planning	path	c-space	known	N/A	perfect
Lazy classical motion planning	path	c-space	known	arbitrary	perfect
Coverage	path	i-space	known	N/A	perfect
Exploration	*	i-space	unknown static	*	*
Navigation	*	c-space	unknown static	*	*
VAMP	path	c-space	unknown static	directional	perfect
Canadian Traveller	path	c-space	unknown static	vertex incidence	perfect
Dynamic Window Approach	controller	N/A	unknown dynamic	directional	
Bug algorithms	controller	c-space	unknown static	ball	perfect
Perception-Aware Motion Planning	path	*	known	N/A	limited

watchman suffices, a solution is a path, which the pursuer follows, that guarantees that any evaders will be detected. The evaders are generally assumed to move arbitrarily fast, but cannot teleport. Solution paths to this problem necessarily cover the whole space, and additionally must also ensure that evaders cannot enter a region that was previously inspected if the region will remain out of sight. In general the paths must revisit the same robot configuration with different information states.

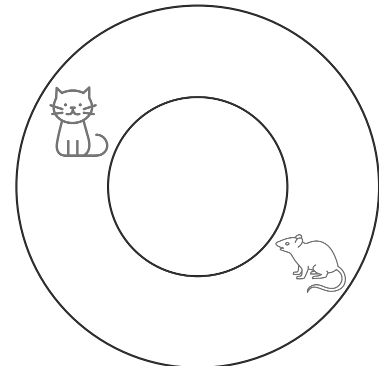


Figure 2.4: An environment in which there is no solution for a single *pur*suer to catch an evader.

When coverage achieved by following a path, the problem setting of covering a target set with a subset of arbitrary covering sets is often called *set cover* [20]. When the covering sets are geometric in some sense (boxes, half planes, balls, etc.), the setting is called *Geometric Set Cover* [21]. If the covering sets are further constrained to correspond to visibility (e.g. in planar problems, the sets are visibility polygons), then determining the set of viewpoints (i.e. stationary watchmen) to cover a target is called the *Art Gallery Problem* [22].

Exploration problems

Exploration problems generally assume no prior knowledge of the obstacles and desire that the robot eventually observe all parts of the space, possibly while building a map of the obstacles [10, 23–28]. In computational geometry, the "online polygon exploration" problem [29] is such an example. Many computational geometry treatments of both coverage and exploration problems assume that coverage occurs continuously as the agent moves Some versions of the problem use "time-discrete vision", which is an important consideration for robots that must be stationary while an observation is collected [30].

Unless otherwise noted, exploration strategies generally assume perfect robot localization. *Simultaneous Localization and Mapping* (SLAM) is usually framed as an estimation problem. "Active SLAM" refers to an approach that decides actions to execute for the sake of exploration, without assuming perfect localization [31].

Navigation problems

Navigation problems seek to reach a specified configuration goal region in an incompletely known environment [32–34]. The general problem of optimizing a path between two vertices in a static, deterministic graph in which the edge weights are not known a priori is known as the *Canadian Traveler Problem*, which is discussed in more detail below.

Connection to lazy search

It is worth noting that lazy motion planning approaches [35, 36], which attempt to reduce the time complexity of motion planning by postponing collision checks, fall into this category of planning on unknown graphs. All of the edge weights are "known" a priori, in the sense that they are computable by the planner — no sensing is required to measure them — but since computing this information is often time-consuming, lazy motion planning strategies generate tentative plans using incomplete information, and decide on a strategy for evaluating the graph edge weights. These approaches differ conceptually from navigation problems, however, since during execution of a navigation plan, the robot is restricted to a certain sensing model, and generally cannot evaluate arbitrary edges.

Determining traversability of graph edges

In the graph-theoretic formulation of motion planning, vertices are embedded in the configuration space, and edges correspond to motion primitives [35]. When all edges in the graph are traversable, then any path in the graph is a valid, executable path. Otherwise, either sensing or computation must occur to *determine* the traversability of an edge.

In lazy motion planning, collision checking computation is deferred in the interest of reducing overall computation time [36]. A planner may speculatively generate successors of a configuration, betting on the reachability of that configuration. This strategy has proven to be effective in many diverse contexts in computing [37]. Lazy motion planning strategies must generate tentative plans using incomplete information, and decide on a strategy for evaluating the graph edge weights. In this setting, the traversability of any edge can be determined, without any commitment to executing a robot motion.

In the *Canadian Traveler Problem* [38, 39] and the *Blindfolded Traveler Problem* [40], the traversability of an edge can be determined once the robot is at a vertex incident to the edge (formulations with "remote sensing" have been considered [41]). This is the same "graph discovery" model as D* [42], but, whereas D* assumes that all edges are traversable optimistically ("optimism in the face of uncertainty"), these methods enable less restrictive assumptions on traversability, e.g. using probabilities, and the planning objective may involve maximizing the probability of success.

CTP and BTP differ only in the formulations of cost. In BTP, an edge is detected as blocked by attempting to traverse it, and there is a cost incurred for partially traversing the edge up until the point where the obstacle is detected. Traditional shortest-path planning, CTP, and BTP can be harmonized by allowing actions that attempt traversing blocked edges, and describing the sensing in terms of ∞-, 1-, and 0-lookahead, respectively. [40]

It is worth remembering to distinguish between planning and execution, or the planner and a controller. While these planners may make an optimistic assumption that an edge is traversable, the overall problem formulation does not. The problem formulations instead determine at what point the traversability of an edge can be determined.

For VAMP, in contrast, determining whether an edge is traversable requires satisfying a more complicated condition. Stated graph-theoretically, each edge induces a disjunction of conjunctions of vertices that must be visited before an edge can be determined. For example, the domain in Figure 1.5 is constructed so that any edge that leads to the goal induces a disjunction of conjunctions where each conjunction contains at least two vertices. Namely, suppose the configurations q_2 and q_3 have nearby configurations q'_2 and q'_3, respectively, that achieve the same visibility, and that there are no other such configurations (for the sake of a small example). The edge leading to q_4 is traversable if and only if

$$(q_2 \land q_3) \lor (q'_2 \land q_3) \lor (q_2 \land q'_3) \lor (q'_2 \land q'_3)\,,$$

where through abuse of notation, a configuration q_i is a truth value indicating whether it has been visited.

Planning for perception

When there is non-negligible actuation and localization uncertainty, the assumptions of traditional motion planning are untenable. If there are stable landmarks in the environment, the motion planning problem can involve reasoning about landmarks. Similarly, visual odometry might rely on regions of the scene that have salient properties (like sufficient texture). Much work addressing robot state uncertainty assumes that the environment is known, and therefore does not apply to the VAMP setting. Paths may be constrained to keep a stationary landmark visible throughout [43] (this work assumes an environment with no obstacles). *Perception-Aware Path Planning* [44] has an objective favoring paths where high-texture regions are in view of the robot-mounted camera so that localization accuracy during execution (which depends on visual odometry) does not degrade. *Coastal navigation* [45, 46] has an objective favoring paths that travel closer to walls instead of through featureless regions of the environment to the same effect. The planning state space may be augmented (in addition to the robot configuration) to include localization uncertainty; some formulations in principle may permit anisotropic sensing for the purposes of localization, but in practice do assume ball visibility [46, 47]. Some formulations rely on smoothness assumptions with respect to sensing that are generally incompatible with visibility-based localization [48–50]. However, none of these methods involves a constraint to steer a sensor for the sake of avoiding unknown obstacles. Some work relaxes the assumption of a known environment, and plans feasible trajectories in a belief space over both robot pose and landmark pose [51, 52]. This formulation in principle may additionally represent belief over obstacle maps, but does not, and refers to another formulation of planning under localization uncertainty to avoid obstacles in an unknown environment, which, again, explicitly assumes ball visibility [53].

The term Simultaneous Mapping and Planning and Simultaneous Localization and Planning has been used in analogy to Simultaneous Localization and Mapping [50, 54].

2.3 Assumptions

In this section we examine some assumptions commonly made in robot systems.

Ball of visibility

Formulations in which the robot geometry is a single point (or a grid cell, for discrete-space formulations) in the workspace often

have an implicit assumption that the robot has visibility of a ball in configuration space. This assumption may be justified for an uncrowded environment and for a robot that is small relative to the obstacles. In this setting, at least with respect to the obstacle map representation, the robot does not have an orientation (or any other degrees of freedom); the configuration space and the work space are equivalent to each other. Such is the case in the heuristic incremental search for the navigation algorithm D* [42], along with numerous other settings. In frontier-based exploration [23], a *frontier* is a region on the boundary between the unmapped regions and known-free regions of the work space. It is often stated that a robot must "navigate to a frontier point", but a frontier, a work space notion, must be *injected* into the configuration space, in order to generate a navigation goal. Carefully stated, the navigation problem is to find a path to a configuration that observes some unmapped region of the work space, where the path of the robot must stay within the region of work space that is known to be free. This navigation subproblem is a subproblem of some of our algorithms for VAMP.

The notion of a ball of visibility in configuration space applies beyond mobile navigation. In an extension of Lumelsky's Bug algorithms to manipulator planning, the assumption is physically realized: "'Clearly, if any point of the arm body is subject to collision, the only way to guarantee obstacle detection is to supply every point of the body with the sensing capability. In this study, the surface of the robot arm is covered with sensitive skin." [55].

In some approaches, safety with respect to unknown obstacles is achieved by constructing a set of "primitive motions that allows the [robot] to turn on the spot, and move within the camera's field of view" [24], thereby reducing the planning problem to the case of ball visibility.

In work on designing a controller to follow a planned trajectory while performing collision avoidance, the dynamic window approach, the approach is "implemented and tested using the robot RHINO, which is a synchro-drive robot currently equipped with a ring of 24 Polaroid ultrasonic sensors, 56 infrared detectors, and a stereo camera system. Because the main beam width of an ultrasonic transducer is approximately 15", the whole 360" area surrounding the robot can be measured with one sweep of all sensors." [56].

Safety criteria

Within problems with obstacle uncertainty, an additional source of variation is the notion of *safety*: one might wish to guarantee safety (non-collision with any obstacle) absolutely, e.g. [8, 10], or with high probability with respect to a distribution over obstacles, e.g. [5, 34] and with respect to obstacles that may move, e.g. [57].

Observation model

Whether for navigation or exploration problems, formulations vary in their assumptions about observations. Sometimes an observation is made from some or all states of the robot during execution; the observation depends on the underlying true world map as well as on the robot's state and may be an arbitrary function of those inputs. Typically, observations are assumed to be in the plane and take the form of either a fixed cone or a circle centered on the robot's location, although more general 3D observations have been considered [26]. As previously mentioned, the robot is typically assumed to be small relative to the obstacles and the environment uncrowded, so that the robot can be approximated as a point. These simplifying assumptions blur the distinction between work space and configuration space and limit the application of these algorithms to more general settings, such as those arising in mobile manipulation. In many formulations of navigation problems, it is taken for granted that traversability information will be readily acquired by the robot.

Motion model

In much of the related work, robot motion is typically assumed to be planar. However, during mobile manipulation, all the degrees of freedom of the robot may affect observations, e.g. the arms may partially block the sensor Figure 1.2. This is the exact setting motivates our work. Previous work has considered a variety of motion models: kinematic, whether holonomic or non-holonomic [27], or kinodynamic [7, 32], and with deterministic or noisy actuation [28, 33]. Robot dynamics introduce additional difficulty because the robot is not able to stop instantly when an obstacle is detected; instead, it must ensure that it never enters an *inevitable collision state*(ICS) with respect to its current known free space and a motion model of possible obstacles.

Environment model

There are several forms of assumptions that can be placed on the environment. The environment can be assumed to be static, dynamic, consist of structured obstacles (with unknown poses, or parameters), or unstructured clutter. In practical deployments, there must be care in updating a map. If objects don't persist long enough, the planner can thrash back and forth. If they persist too long, no plan may be found, even though the objects have been (re)moved.

Search and execution strategies

Problems dealing with map uncertainty (navigation, lazy search, or exploration problems) typically involve solving similar search problems as information is incrementally gathered. Problems with map uncertainty, as in our case, can all be cast as some version of a partially observed Markov decision process (POMDP). For our version of the problem, the *state space* would be the cross product of the robot's configuration space with the space of all possible arrangements of free/occupied space defined by the unknown obstacles, the actions would be finite linear motions in configuration space, and the observations would be determined by the visibility function of the sensor. The objective would be to minimize an objective involving path length and an infinite penalty for moving into obstacles and outside the visible region. Seeing the problem this way is often clarifying, but it does not immediately lead to a solution strategy, since the optimal solution of POMDPs is highly computationally intractable even in discrete state and action spaces. If it were possible to solve the POMDP optimally, the result would be a policy that maps the robot configuration and a distribution over (or set of) possible maps consistent with the history of observations into actions. Such a policy is intractable to explicitly compute or even represent.

Practical approximation strategies for POMDPs almost all rely on some form of receding-horizon control [10]. The system makes a plan for a sub-problem under some assumptions, and begins executing it, gathering further information about the map as it goes. When it receives an observation that invalidates its plan (e.g. an object in its path) or reaches its subgoal, the system makes a new plan based on the current robot configuration and map state.

One example of this approximation strategy is in exploration problems, where a typical strategy is some form of *frontier-based* or *next-best view planning* method [23, 26, 27]. On each replanning

iteration, a subgoal configuration is selected (a) that is reachable within known free space from the robot's current configuration and (b) from which some previously-unobserved parts of the work space can be viewed. A motion planner appropriate for the robot's dynamics is used to plan a path to that subgoal configuration, while staying within the previously observed space, possibly with an additional objective of viewing as much unobserved work space as possible along the way; the robot executes the path, gathering information, and then replans. The most important question in these methods is how to select subgoals. Even if the information utility is submodular [58], a greedy strategy may be significantly suboptimal in terms of robot motion, as it can cause the robot to move back and forth between distant subgoals.

When the objective is navigation, a typical replanning strategy is to be optimistic, planning a path to the goal that makes some assumptions about the true map and replanning if that assumption is invalidated. There are several approaches to reusing computation for searching on a modified graph [59]. It is generally the job of an execution monitor (e.g. [60]) to update the graph based on observations. Our main contribution in this thesis addresses the planning component of this approach.

3 Path Search

We will ultimately focus on path search as it applies to VAMP, but we can develop some concepts abstractly, without any geometry, by considering graphs where each edge is labeled with a set of vertices that must be visited before the edge is traversable. A path from one vertex to another might have to make an excursion to visit other vertices, which resembles the situation of long-range dependencies in VAMP domains. For general graphs, it is easy to show that this problem is NP-hard.

3.1 Introduction

We will speak of a "planning problem", and algorithms for solving it. For a planning instance, we have a set of solutions. If we have an optimizing planning problem, we distinguish between feasible solutions and optimal solutions.

Given an algorithm, a planning instance, and possibly some initial algorithm state (e.g. the seed of a pseudorandom number generator), we can produce a result.

An algorithm is "sound" if all results produced by it solve the planning problem. An algorithm is "complete" if given a planning problem with a solution, the algorithm is guaranteed to produce a valid solution. The "completeness" property can be with respect to either feasibility or optimality.

In a typical formulation, to design a solution strategy based on search for a planning problem, a designer identifies a state space, and expresses transition costs, and the goal test in terms of the state space. The valid transitions from a state are called the successors of a state. Instead of states, we can think of *search nodes*. A search node contains a reference to its predecessor.

It can be convenient to consider all transitions to be valid, but those that are invalid in spirit incur infinite cost.

Predecessor and *parent* are synonyms. The reference to the predecessor is sometimes called a *back pointer*.

A state summarizes the relevant information in a path and initial state. The path may be over configurations, or it may be a path of actions if it is necessary to distinguish different paths between configurations (e.g. if a path contains two successive configurations that are a half-turn apart, it can be unclear which direction the robot turned) or if determining the action given two configurations is not trivial (e.g. requires solving a two-point boundary value problem).

Typically, the cost incurred in achieving a state is formulated to be exogenous to the state itself. If, for example, cost corresponds to energy, and certain actions may be unavailable if there is not enough energy, this notion of cost must be represented in the state.

This formulation is extremely flexible and useful. In a setting with finitely many actions, it is possible to use a priority queue to organize the search process. The priority queue maintains (at least in part) the "algorithmic state" of the search process. Within this formulation, there are possible variations of data layout. For our sake, we will say that an element of the priority queue is a pair of type (node, priority). There is a start node, and to initialize the algorithm, the state is placed on a queue with an arbitrary priority. On every iteration, a minimal node is extracted from the queue. A procedure generates successors for that node, computes corresponding priorities, and inserts successor nodes into the queue. This procedure typically iterates until a goal node is extracted from the priority queue, until the priority queue is empty, or until a timeout is reached. These outcomes correspond respectively to a problem that is feasible, infeasible, or indeterminate.

The qualifier "algorithmic" serves to distinguish from the typical notion of state in a search problem. It is sometimes referred to as the "metalevel state space" [61].

In practice, it is almost always necessary to *prune* nodes, e.g. those representing multiple paths to the same state, either before inserting them into the priority queue or before generating successors.

We adopt a slightly different view of search. A search node is nothing more than an efficient representation of a path. Fundamentally, the priority queue stores paths. To make the discussion more concrete, we will consider walks on a square lattice graph of configurations of size (m, n).

A more common perspective is that the priority queue stores states, and there is additional bookkeeping keep track of the predecessor of states in the priority queue. By the definition of the state space, there is no need to keep track of multiple predecessors for the same state.

We use $\mathscr{C} = \{(i, j) \mid i \in [1, m], j \in [1, n]\}$. We will consider walks starting at vertex $(1, 1)$ and ending at (m, n), where each step changes one coordinate by one unit (i.e. 4-connected). The goal configuration is (m, n), and in our search formulation goals are "absorbing", so we do not consider walks that visit (m, n) in the middle. Let $W_{(m,n)}$ stand for the set of all such walks. $W_{(m,n)}$ is an infinite set since a walk may contain cycles, e.g. $[(1, 1), (1, 2), (1, 1), (1, 2), \cdots, (1, 1), (1, 2), (2, 2)] \in W_{(2,2)}$. We will introduce subsets of $W_{(m,n)}$ that are finite, and use our search formulation to explicitly enumerate all the walks in these subsets. It is uncommon in robotic applications of path search to require enumerating multiple equivalent paths that solve a search problem. It is more common to find a single solution. If that solution is inadequate, then search problem is updated such that the previously generated path is invalid. The number of possible paths in the lattice does not necessarily correspond to the difficulty of finding a specific solution path. We are most interested in what we call *informative walks*. The number of informative walks on an (m, n)-lattice does not appear to be published (for example, it is not yet in the OEIS [62]). Because integer sequences are so distinctive

We use the term "walk" because some authors use "path" to mean a "walk" where all the vertices are unique. Later we will use "path" to mean "walk".

and characteristic, yet so abstract, documenting even the first few terms might lead to interesting connections.

3.2 Pruning without affecting correctness

If the aim is to produce a cost-minimal path, then in the presence of duplicate states, the lower-cost one is kept. There are two distinct operations: determining a notion of equivalence of nodes, and a comparison of cost. The process of removing paths from the consideration of a search procedure is called "pruning". If pruning is performed on the basis of state equality, then it does not affect the correctness of the search algorithm. On a regular lattice, there are many ties, and this pruning can exponentially reduce the number of paths to be considered.

If the heuristic is *consistent*, then the first path found to a state is guaranteed to be a cost-minimal path. When the heuristic is merely *admissible*, then an optimal search must consider additional paths to reached states. [61]

The term *pruning* usually does not apply if there is a static set of paths that are removed, such as paths that collide with an obstacle. Such a constraint expressible in the successor function.

3.3 Generalizing pruning

We define a projection of the state space into a space on which the equivalence check is performed. We can recover the traditional formulation by using an identity projection. The advantage of interest is that we have additional flexibility for defining pruning rules that serve to approximate complete searches. The choice of pruning is not tied to the choice of state space. By isolating path pruning into this projection and domination, we can describe different search problems in a unified manner. A secondary benefit to this alternate formulation is that there is a small change in the algorithm between finding a single shortest path and finding *all* shortest paths. To find a single shortest path, the search prunes any paths that are equal or worse in cost. To find all of them, the search prunes any path that is strictly worse in cost.

3.4 Shortest paths

Consider all the shortest paths in $W_{(m,n)}$. If our goal is to count the paths, there is a straightforward combinatorial argument. Any shortest path must take m steps in the first coordinate (rightward) and take n steps in the second coordinate (upward). Any interleaving of these lattice steps will produce a shortest path, and so there are $\binom{n+m}{n}$ paths.

To enumerate these paths in our formulation, we specify

- a successor function where a path of length l generates all possible successor paths of length $l + 1$ (all lattice moves are valid),
- a path projects to its tip configuration,
- the domination quantity is the length of a path, and
- a pruning rule that prunes paths that are longer than the minimum domination quantity found so far.

We use the goal test to keep track of the enumerated paths, but we keep searching until the priority queue is empty. We also use the goal to "absorb" paths. If an extracted path satisfies the goal, its successors are not generated, and they are not inserted into the queue.

We could remove successors that are guaranteed not to be on a shortest path, such as successors that revisit a configuration.

This point is inconsequential for shortest paths (), but not so for self-avoiding walks.

We enumerate paths with our implementation of path-dependent search (Algorithm 1) and verified the number of paths for modest lattice sizes against the combinatorial argument above.

3.5 Self-avoiding walks

Consider all the self-avoiding walks in $W_{(m,n)}$. A walk is self-avoiding if there are no repeated vertices in the walk. We encode this constraint in the successor function; only lattice moves that self-avoid the path so far are valid.

Any shortest path is also self-avoiding.

Here, v can stand for “visited”. In Chapter 4, v will stand for “visible”, and will be a subset of a distinct space $\mathscr{W}$ not $\mathscr{C}$.

To find *one* self-avoiding walk in the state-space formulation, the state space is isomorphic to $\mathscr{C} \times \mathbb{P}(\mathscr{C})$. An element is (q, v) with $q \in \mathscr{C}$ and $v \subseteq \mathscr{C}$. If the aim is to generate all self-avoiding walks in this space, we disable pruning. See Figure 3.1 for two paths that must be distinguished.

We enumerate paths with our implementation of path-dependent search (Algorithm 1) and verified the number of paths for modest lattice sizes against OEISA064298 [63].

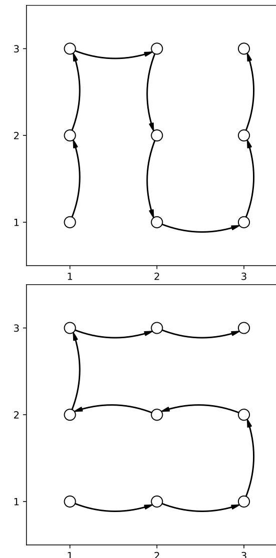


Figure 3.1: Two distinct walks (”N”and “S” shapes) with the same start and end configurations, visiting the same configurations along the way.

3.6 Informative walks

The self-avoiding constraint above is one example of a constraint that can be expressed in a (q, v) state space. We consider now a validity check that eliminates cycles in the (q, v) space mentioned above. We call these walks “informative” because we imagine v to represent some information, and some information is necessary before a transition can be taken. For example, consider the path $p = [q_1, q_2, q_1, q_2]$. Lifted into the (q, v) space, the path is $\text{lift}(p) =$

$[(q_1, \{q_1\}), (q_2, \{q_1, q_2\}), (q_1, \{q_1, q_2\}), (q_2, \{q_1, q_2\})]$. Revisiting q_1 at $pl[3]$ is fine, because there is new information, and perhaps a new transition is now valid at q_1. However, lift(p)[4] = lift(p)[2]. There is nothing possible from $pl[4]$ that was not possible from $pl[2]$. A walk p is informative if and only if lift(p) does not contain any repeated elements.

Any path containing p as a subpath is redundant. The path $[\cdots, q_1, q_2, q_1, q_2, \cdots]$ is redundant since the cycle could be removed to form the equivalent path $[\cdots, q_1, q_2, \cdots]$

For the $(2, 2)$ lattice, with nodes labeled $(1, 1), (1, 2), (2, 1), (2, 2)$ there are 6 paths that do not cycle in (q, v) space:

- 1: [(1, 1), (1, 2), (2, 2)]
- 2: [(1, 1), (2, 1), (2, 2)]
- 3: [(1, 1), (2, 1), (1, 1), (1, 2), (2, 2)]
- 4: [(1, 1), (1, 2), (1, 1), (2, 1), (2, 2)]
- 5: [(1, 1), (1, 2), (1, 1), (2, 1), (1, 1), (1, 2), (2, 2)]
- 6: [(1, 1), (2, 1), (1, 1), (1, 2), (1, 1), (2, 1), (2, 2)]

3.7 Vertex visit graphs

We now introduce in Figure 3.2 a graphical tool that will play two roles. We call such a graph a *vertex-visit graph*. We will first use vertex visit graphs to illustrate the vertex visit sets between configurations of a given walk. We will then consider vertex visit graphs as an abstract version of VAMP where there is no geometry.

On the $(3, 3)$ lattice, there are 242220 informative paths. We provide a summary of quantities we were able to compute with our implementation in Table 3.1

On the $(3, 3)$ lattice, we observe that there are informative paths that induce the same vertex visit graph. In fact, there are 182932 unique vertex visit graphs generated by the 242220 paths. We will look in detail at two paths that produce the vertex visit graph in Figure 3.3. Consider three segments

lattice	informative paths
$(2, 2)$	6
$(2, 3)$	98
$(2, 4)$	3909
$(2, 5)$	334276
$(2, 6)$	60230715
$(3, 3)$	242220
$(3, 4)$	> 95 million

Table 3.1: The counts of informative paths for a few modest lattice sizes.

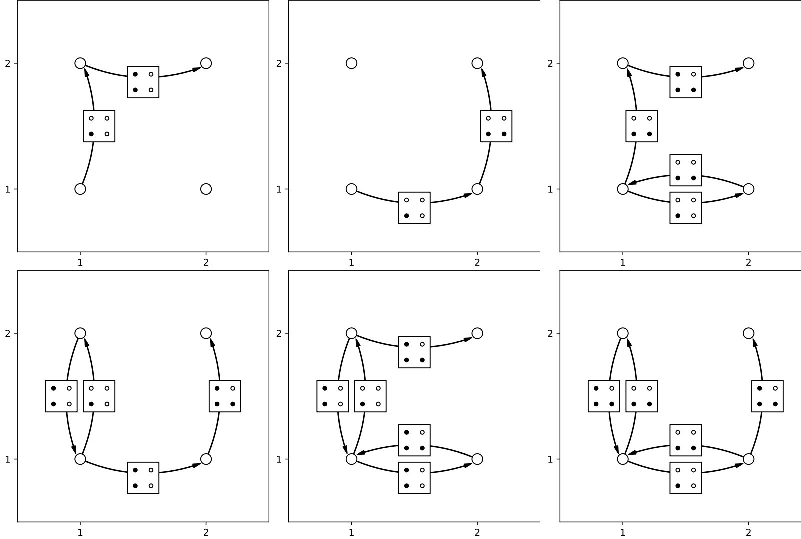


Figure 3.2: Each of the 6 informative paths on the $(2,2)$ lattice induces a graph with edges labeled with subsets of vertices.

- $A = [(1,1),(1,2),(2,2),(1,2),(1,1),(2,1),(2,2),(2,3),(1,3),(1,2)]$, and
- $B = [(1,1),(2,1)]$, and
- $C = [(2,2),(2,3),(3,3)]$.

$A;B$ represents the concatenation of A then B.

The longer path is $A;B;C$, and the shorter path is $A;C$. The longer path visits $(1,1)$ three times. Take a look at these three visits in (q,v) space:

- 1: $((1,1),\{(1,1)\})$
- 2: $((1,1),\{(1,1),(1,2),(2,2)\})$
- 3: $((1,1),\{(1,1),(1,2),(2,2),(2,1),(2,3),(1,3)\})$

There are only two successors from $(1,1)$, both of which are traversed in the A segment. On the third visit, the new information cannot allow any new transitions immediately, since all possible transitions have been taken. We still must consider the path, since the additional information may allow a transition further down the line.

Indeed, each visit of $(1,1)$ contains new vertices, otherwise the path $A;B;C$ would not be informative. The path $A;B;C$ is not redundant on its own. What one might desire to be true is that path $A;B;[(2,2)]$ is *dominated* by $A;[(2,2)]$; they both reach $(2,2)$ with set $\{(1,1),(1,2),(2,2),(2,1),(2,3),(1,3)\}$, but one is longer than the other.

We just saw a case in which one path was longer than the other, but both had the same vertex visit graph. There are also such paths that have the same length. For example, define a common prefix p = [(1, 1), (2, 1), (2, 2), (3, 2), (2, 2), (2, 1), (3, 1), (3, 2), (2, 2), (1, 2), (1, 1), (2, 1)] , The paths $p_1 = p;[(2,2),(3,2),(3,3)]$, and $p_2 = p;[(3,1),(3,2),(3,3)]$ have the same length and induce the same vertex visit graph.

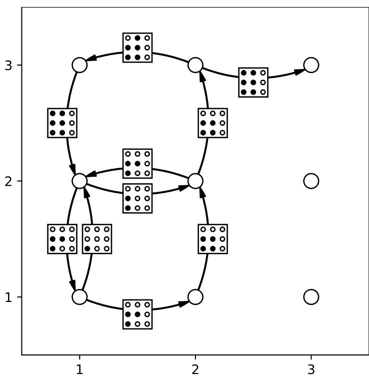


Figure 3.3: The same vertex visit graph can be induced by two informative paths. One path is $A;C$, the other is $A;B;C$, for A = [(1, 1), (1, 2), (2, 2), (1, 2), (1, 1), (2, 1), (2, 2), (2, 3), (1, 3), (1, 2)], and B =[(1, 1), (2, 1)], and C =[(2, 2), (2, 3), (3, 3)].

To summarize, the constraint to generate informative walks applies to an individual path. However, the decision to prune a candidate path is based on what paths have already been found by the search.

3.8 Complexity of vertex visit graphs

Vertex visit graphs are a convenient way to explore complexity of path-dependent problems search formulations.

Quadratic scaling of solution length

We provide an argument for the worst-case length of a walk on a vertex visit graph with n vertices. Assume a solution exists and that a walk exists from q_1 to q_n. The worst case is achieved when we must revisit every node we have already visited to make progress toward q_n. For $n = 6$, this looks like $[q_1, \{\}, q_2, \{q_1\}, q_3, \{q_1, q_2\}, q_4, \{q_1, q_2, q_3\}, q_5, \{q_1, q_2, q_3, q_4\}, q_6]$ The configurations contained within curly brackets are the ones that are revisited, and may be revisited in some other order than depicted. Whatever the order, such a walk is informative, and it is possible to construct a vertex visit graph that requires such a walk. The length of such a walk is $n + 1 + 2 + \cdots + (n - 1) = n + n(n - 1)/2$. This quadratic behavior is also possible in VAMP domains (see Section 4.4).

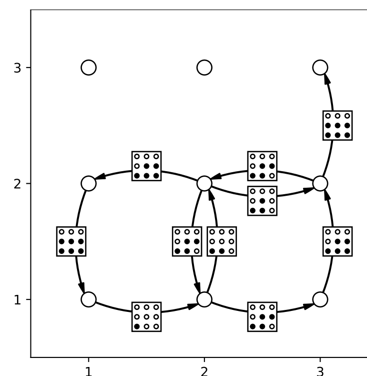


Figure 3.4: The same vertex visit graph can be induced by two informative paths of the same length. One path is $A; B$, the other is $A; C$, for A =[(1, 1), (2, 1), (2, 2), (3, 2), (2, 2), (2, 1), (3, 1), (3, 2), (2, 2), (1, 2), (1, 1), (2, 1)], and

B =[(2, 2), (3, 2), (3, 3)], and

C =[(3, 1), (3, 2), (3, 3)].

The empty brackets are included to help illustrate the pattern.

Polynomial-time feasibility checking for reversible graphs

Consider a vertex visit graph G. G is reversible if and only if

$$[q_1, q_2, \cdots, q_{k-1}, q_k]$$

being a valid walk in G implies that

$$[q_1, q_2, \cdots, q_{k-1}, q_k, q_{k-1}, \cdots, q_2, q_1]$$

is a valid walk in G, too. For such a graph, determining reachability from a source to target vertices can be done in time polynomial in the number of vertices in G. The algorithm in Section 4.2 is described in terms of the VAMP problem, and also applies to finding a walk in a vertex-visit graph.

NP-completeness of vertex visit graphs

The decision problem of determining whether a non-reversible vertex visit graph contains a feasible path from source to target, or whether a reversible vertex visit graph contains a feasible path of length no greater than k is NP-complete. First, the problem is in NP because a solution path is a certificate, and its validity can be

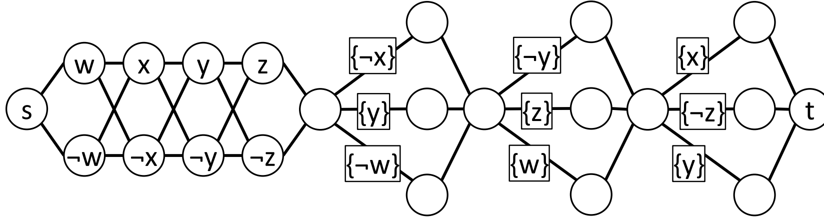


Figure 3.5: An example polynomial-time reduction from 3-SAT to finding a shortest path on a vertex visit graph.

checked in polynomial time. Next, to show the problem is NP-hard, we provide a reduction from boolean satisfiability, specifically 3-SAT. We demonstrate this reduction by example. Take the following 3-SAT problem:

$$(x \vee y \vee w) \wedge (y \vee z \vee w) \wedge (x \vee z \vee y)\,. \tag{3.1}$$

We can encode this as a vertex visit graph with source vertex s and target vertex t, depicted in Section 3.8, where each edge has unit weight. The path first goes through vertices labeled with assignments to the variables (e.g. x and x). After the variables have been assigned, the path goes through the clause gadgets where the fan-out edges are labeled with the vertex that must have been visited to allow the edge to be traversed. All other edges in the graph do not have constraints. If the edges are undirected, as shown, then this is a reversible vertex visit graph, and a feasible path can be found in polynomial time. A feasible path, however, might visit both x and x, which destroys the interpretation of the first part of the path as being an assignment to the variables, since now x is both true and false. If the edges are directed to move rightward, this is not possible.

Similarly, a path that visits x or x, but not both, is shorter than the path that visits both x and x. The formula is satisfiable if and only if there is a path of length (no greater than) 11. Since the graph is polynomial in the size of the satisfiability formula, the reduction can be performed in polynomial time. Colloquially, finding optimal paths in vertex visit graphs is therefore NP-hard. Since we already showed it was in NP, therefore it is NP-complete.

3.9 Configuration path-dependent search

In problems where there is path-dependency on the history of configurations, one typically identifies a state space, and in this state space there is no path dependence. Here, we do not introduce a state space and operate directly in the space of path histories.

Algorithm

We now outline a general search algorithm suitable for path-dependent formulations. Algorithm 1 presents the general structure of a path-dependent search algorithm. DominationTest allows the search to prune certain paths. To maintain a search procedure that is complete, the search can only prune paths that have equivalent path summaries. However, in our application, we use aggressive pruning strategies for efficiency, and rely on a higher-level search procedure to maintain correctness of the overall path. When used in this way, DominationTest has the effect of *aliasing* a set of search states into one representation in the priority queue. In practice, the algorithm would maintain the $visited$ data structure as an associative array indexed by this search-state aliasing key. For problems in which the robot configuration is a "sufficient statistic', e.g. shortest path problems, the path summary is just the last configuration on a path. DominationTest can be chosen to recover the behavior of Dijkstra's algorithm by considering only the first path to a configuration.

Configuration path-dependent problems

In this section, we frame existing problems in this path-dependent formulation. In all of these problems, there's a robot *configuration* and there is a configuration *path summary*. All of these problems require a search over path summaries. The path summary is the information, in addition to the current configuration, to form a state space for the purpose of planning. There are multiple ways in which the formulation may require the path dependency. The cost function on a path may not be decomposable. In this case, we clarify this by considering two paths p_1 and p_2 that can be concatenated, and describing what equality fails to hold. When the cost function is a traditional notion of path length, it does decompose: length(p_1)) + length(p_2)) = length($p_1; p_2$)). Or perhaps the cost function does decompose, but the goal condition depends on the configuration path taken.

The test DominationTest$(p, \tilde{p})$ should return true if $\tilde{p}$ cannot lead to a better solution than p. Typically, $\tilde{p}$ can be pruned if $\exists p$ where $p[\text{end}] = \tilde{p}[\text{end}]$ (both paths lead to the same configuration), and an additional problem-specific condition, shown below.

Risk-Aware Motion Planning [3]

path summary: how much risk is accumulated. $\text{risk}(p) \in \mathbb{R}$.
$\textsc{DominationTest}(p, \tilde{p}) \equiv (\text{length}(\tilde{p}) > \text{length}(p)) \wedge (\text{risk}(\tilde{p}) > \text{risk}(p))$
path dependency: $\text{risk}(p_1) + \text{risk}(p_2) \neq \text{risk}(p_1; p_2)$

Minimum-Constraint Removal [4]

path summary: which obstacles have been violated. $\text{ViolObst}(p) \subseteq O$
$\textsc{DominationTest}(p, \tilde{p}) \equiv \text{ViolObst}(\tilde{p}) \supseteq \text{ViolObst}(p)$
path dependency: $|\text{ViolObst}(p_1)| + |\text{ViolObst}(p_2)| \neq |\text{ViolObst}(p_1; p_2)|$
Note that the problem of minimum swept-volume motion planning can be seen as a version of minimum constraint removal.

See irreducible constraint removal in [4].

Minimum-Risk Motion Planning with Obstacle Uncertainty [64]

path summary: depends on the parameterization of the distribution of the environment, and whether a union approximation is taken. For simplicity, assume environment consists of balls B_i, $i = 1, \cdots, n$ with larger radii being less probable. $\text{risk}(p) \in \mathbb{R}^n$, where n is the number of obstacles
$\textsc{DominationTest}(p, \tilde{p}) \equiv \text{risk}(\tilde{p}, B_i) > \text{risk}(p, B_i) \forall B_i$
path dependency: $\text{risk}(p_1, B_i) + \text{risk}(p_2, B_i) \neq \text{risk}(p_1; p_2, B_i)$

Optimal Visibility-Based Pursuit Evasion [6]

path summary: status of shadows, contaminated or uncontaminated
$\textsc{DominationTest}(p, \tilde{p}) \equiv (\text{length}(\tilde{p}) > \text{length}(p)) \wedge ()\text{UncontaminatedShadows}(\tilde{p}) \subseteq \text{UncontaminatedShadows}(p))$
path dependency: $\textsc{GoalTest}(p) \equiv \text{ContaminatedShadows}(p) = \emptyset$
note: Optimal path may revisit the same configuration.

See Section 8.4. Pruning via path dominance in [6].

Belief-Space Planning [7]

path summary: covariance matrix, assuming some distribution of observations at planning time In this case, $p[\text{end}]$ represents the mean and variance under some distribution of observations. And we can generate confidence intervals from the distribution over obstacles.

$\textsc{DominationTest} p, \tilde{p} \equiv \text{conf_int}(\text{var}(p), \sigma) \subseteq \text{conf_int}(\text{var}(\tilde{p}), \sigma) \forall \sigma$

This is a centered ellipsoid inclusion test, equivalent to a positive definite test on the difference of covariance matrices. In practice, it may be unlikely that different paths achieve exactly the same mean, and so a general ellipsoid inclusion test could be used.

path dependency: path constraints (via chance constraint), and goal test

For VAMP, the path summary is the set of previously visited configurations, plus initial visibility. Or $\mathbb{P}(W)$. The aspect of the search problem that is sensitive to the path summary is the extension check, via Equation V-Safety Constraint as the *visibility constraint*. In the first three problems, the path dependence does not play a role in determining path legality.

Algorithm 1

1: **procedure** PathDepSearch(G, $path_{start}$, ValidTest, QueueOrder, DominationTest, GoalTest)
2: $queue \leftarrow PriorityQueue()$
3: $queue \leftarrow$ Push($queue, path_{start}, 0$))
4: $visted \leftarrow \emptyset$
5: $best_path \leftarrow$ **null**
6: $i \leftarrow 0$
7: **while** $\|queue\| > 0$ **do**
8: $i \leftarrow i + 1$
9: $(queue, path) \leftarrow$ PopMin($queue$)
10: **if** GoalTest($path$) **then**
11: **return** $path$
12: **end if**
13: **for** $path_{old} \in visited$ **do**
14: ▹ the domination test typically requires some sort of equality to pass, so $visited$ can be an associative array and iterate only on a subset.
15: **if** DominationTest($path_{old}, path$) **then continue**
16: **end if**
17: **end for**
18: **for** $c \in$ Descendents($G, path$[end]) **do**
19: $path_{new} \leftarrow path; [c]$
20: **if** ValidTest($path_{new}$) **then**
21: $queue \leftarrow$ Push($queue, path_{new}$), QueueOrder($path_{new}$)
22: **end if**
23: **end for**
24: $visited \leftarrow visited \cup \{path\}$
25: **end while**
26: **return** $best_path$
27: **end procedure**

4 Formulation and Algorithms

In this chapter we present the core contribution of this thesis: a general formulation of VAMP and algorithms for producing solutions to VAMP instances.

4.1 Formulation

We formally state the VAMP problem as a generalization of traditional motion planning, with additional constraints. We also provide a generalization that relaxes the constraint, and introduces a workspace visibility goal because it arises in one of our solution approaches of the original problem.

A traditional motion planning formulation may be a tuple consisting of

Configuration Space	$\mathscr{C}$
Work Space	$\mathscr{W}$
$\mathscr{C}$ Neighborhood	$N(q) \subseteq \mathscr{C}, q \in \mathscr{C}$
Collider Region	$\text{col} \subseteq \mathscr{W}$
Swept Region	$S(q_1, q_2) \subseteq \mathscr{W}, q_i \in \mathscr{C}$
Initial Configuration	$q_{\text{init}} \in \mathscr{C}$
Goal Configuration Set	$Q_{\text{goal}} \subseteq \mathscr{C}$

We focus on problems in which the robot motion is quasi-static, so the state of the robot can be modeled entirely by its configuration. In this setting, we assume some set of motions giving the connectivity in configuration space, e.g. a probabilistic roadmap [65] or a regular lattice graph embedded in configuration space. Therefore, $\mathscr{C}$ is a finite set, and the problem instance includes an implementation of a connectivity function N. This function encodes a topology of the configuration space and does not encode feasibility with regard to collisions. $S(q_1, q_2)$ is the region swept by the robot when following a primitive motion from q_1 to q_2. The motion from q_1 to q_2 is topologically valid and feasible when $q_2 \in N(q_1)$ $S(q_1, q_2) \cap \text{col} = \emptyset$. $S(q) = S(q, q)$ is used to denote the region occupied by a robot in configuration q.

It is also helpful if the robot is holonomic, because the motion primitives can be straight line segments in configuration space.

It is common to approximate $S(q_1, q_2)$ by sampling configurations from q_1 to q_2, possibly just those two endpoints, in contrast to doing continuous collision checking. In practice this is possible when N allows only small steps in configuration space, along with using an overapproximation of $S(q)$, such that the discrepancy caused by the approximation is negligible.

A solution to this motion planning problem is a sequence of configurations that starts at q_{init}, ends in Q_{goal}, is connected according to N and is feasible with respect to S and col. We may additionally be interested in some cost function on the path, which almost

always decomposes additively, such as the length of the path in configuration space.

An example of a non-additive cost function is to minimize the total swept volume of the path.

To arrive at our first definition of a VAMP problem instance, we extend the above table with additional entries

Occluder Region	$\text{occ} \subseteq \mathcal{W}$
Visible Region	$V(q) \subseteq \mathcal{W}, q \in \mathcal{C}$
Initial Visible Set	$v_{\text{init}} \subseteq \mathcal{W}$
Goal Configuration Set	$Q_{\text{goal}} \subseteq \mathcal{C}$

$\mathbb{P}(A)$ is the *power set* of A, the set of all subsets.

In most cases, col = occ, but we nevertheless make the distinction. In general, there are transparent obstacles (e.g. glass) and opaque non-obstacles (e.g. fog). There may also be a region that does not generate occlusions, but should nevertheless exclude the robot.

$V(q)$ is the region that is visible by the robot at q, taking into consideration occ and self-occlusions.

We are assuming that motion is continuous-time (so all configurations along a path must be previously viewed) but that perception is discrete-time (so new views are only gained at the end of each primitive trajectory). In general, $S(q_{\text{init}}) \nsubseteq V(q_{\text{init}})$, and so to admit a solution, v_{init} must be large enough so that the robot may take some safe action.

Altogether, a VAMP problem instance is a tuple

$$D = \left(\mathcal{C}, \mathcal{W}, N, \text{col}, \text{occ}, S, V, q_{\text{init}}, v_{\text{init}}, Q_{\text{goal}},\right) \tag{4.1}$$

Where it is necessary to discuss multiple problem instances, e.g. D_1 and D_2, we distinguish between fields of different instances with $D_1.v_{\text{init}}$ and $D_2.v_{\text{init}}$.

We now introduce some notation used to characterize paths P:

$$P[i] \in \mathcal{C}, i = 1, \cdots, n\,.$$

For convenience, define $\text{length}(P) = n$ and define $P[\text{end}] = P[\text{length}(P)]$

Define (by overloading notation) a corresponding sequence of accumulated visibility $V(P)$:

$$V(P)[i] \in \mathcal{W}, i = 0, \cdots, n$$

with $V(P)[0] = v_{\text{init}}$ and

$$V(P)[i] = V(P)[i-1] \cup V(P[i])\,, i = 1, \cdots, n\,.$$

cf. prefix "sum", cumulative "sum", or scan

Note that $V(P[i])$ is the subset of $\mathcal{W}$ that is visible at configuration $P[i]$, whereas $V(P)[i]$ is the subset of $\mathcal{W}$ that is visible from a configuration $P[j]$ for any $j = 1 \cdots i$.

Note that this formulation bakes in the assumption that the acquired visibility increases monotonically.

Define a corresponding sequence of incremental violations

$$X(P)[i] \in \mathcal{W}\,, i = 1, \cdots, n\,,$$

with e.g. $X(P)[1] = \emptyset$ and

$$X(P)[i] = S(P[i-1], P[i]) \setminus V(P)[i-1]\,, i = 2, \cdots, n\,.$$

To represent set-theoretic difference, we use $A \setminus B = \{x | x \in A \wedge x \notin B\}$

It can be convenient to refer to sequences with more traditional notation. We will use q_i for an element of sequence of configurations, and v_i for an element of a sequence of accumulated visibility.

$$V([q_1 \cdots, q_n]) = \bigcup_{i=1\cdots n} V(q_i) \cup v_0 \tag{4.2}$$

$$S([q_1 \cdots, q_n]) = \bigcup_{i=1\cdots n-1} S(q_i, q_{i+1}) \tag{4.3}$$

P is a solution to the VAMP instance if it satisfies:

A union of no terms is $\emptyset$, so $V([]) = v_0$ We do not rely on it, but an intersection of no terms is the universe. From the perspective of a computational implementation with an "accumulator" these choices are as natural as choosing zero/one for a sum/product of no terms, since they correspond to the initial value of the accumulator. An *unbiased definition* would not require a definition of these edge cases [66].

$$P[1] = q_{\text{init}} \tag{4.4}$$

$$P[\text{end}] \in Q_{\text{goal}} \tag{4.5}$$

$$S(P[i], P[i+1]) \cap \text{col} = \emptyset\,, i = 1 \cdots n-1 \tag{4.6}$$

$$X(P)[i] = \emptyset\,, i = 1 \cdots n \qquad \text{(V-Safety Constraint)}$$

We refer to Equation V-Safety Constraint as the *visibility constraint* or *safety constraint*. It could alternately be expressed as

$$S(q_i, q_{i+1}) \subseteq V([q_1, \cdots, q_i])\,, \forall i \in \{0, \ldots, n-1\} \tag{4.7}$$

For sets A and B, the statement $A \setminus B = \emptyset$ is equivalent to $B \subseteq A$. We express Equation V-Safety Constraint in terms of set difference because it will be useful to allow violations of the constraint, and to keep track of where those violations occurred.

Figure 4.1 graphically depicts the sequence of sets X for a path that is not safe with respect to Equation V-Safety Constraint.

Because we are interested in the length of solution paths, define $D(q_i, q_j)$ to be the length of the primitive trajectory moving

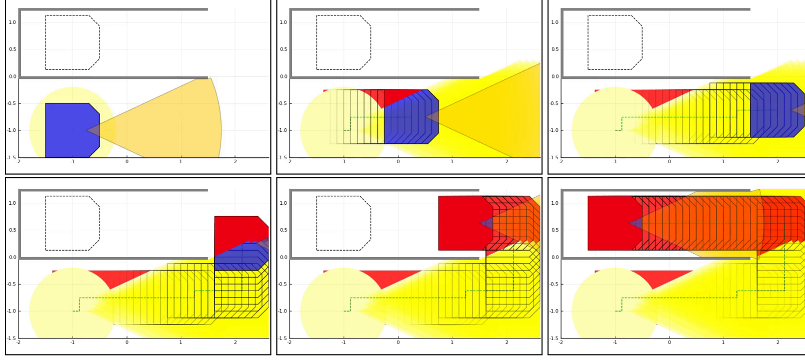


Figure 4.1: Illustration of violation region X shown in red. In the HALLWAYHARD domain, a motion plan that is constrained only to avoid obstacles incurs violations of Equation V-Safety Constraint.

from q_i to q_j. The length of a path is similarly $D([q_1, \cdots q_n]) = \sum_{i=1}^{n-1} D(q_i, q_{i+1})$

Formulation of subproblem

The purpose of VAMP is to plan a path to a goal that is safe with respect to Equation V-Safety Constraint. We have so far taken the goal to be a set in configuration space, and we consider a solution to satisfy the goal if the path terminates with a configuration in the goal set. As we have alluded to, we may relax Equation V-Safety Constraint, which would produce a violation region. We would like to plan a path that acquires visibility of this region, and so we extend our current formulation to include an additional goal:

Goal Visible Set (disjunction) $\quad V_{\text{goal}} \subseteq \mathbb{P}(\mathcal{W})$

A solution path P satisfies this overall VAMP instance if it satisfies

$$\exists v_{\text{goal}} \in V_{\text{goal}}\, V(P)[\text{end}] \supseteq v_{\text{goal}} \tag{4.8}$$

(there exists a set in V_{goal} such that P acquires visibility of that set) in addition to Equation 4.5, Equation 4.6, and Equation V-Safety Constraint.

If there is effectively no goal to acquire visibility, then $V_{\text{goal}} = \mathcal{W}$, which is how we recover the previous formulation. If there is a visibility goal, then in practice $Q_{\text{goal}} = \mathcal{C}$.

4.2 Algorithms for visibility-aware motion planning

We present several algorithmic approaches to the VAMP problem, beginning with a computationally inefficient method that produces

very high quality plans, and then exploring alternative strategies that are more computationally tractable. We identified a strategy for solving VAMP instances based on pruning, constraint relaxation and goal regression.

As formulated in Section 4.1, in all these algorithms, we assume a given finite graph (Q, E) where $Q \subset C$ is a set of configurations and all edges $(q_1, q_2) \in E$ are collision-free with respect to W_{obs}, so $S(q_1, q_2) \cap W_{\text{obs}} = \emptyset$. Any of our VAMP algorithms can be augmented by an outer loop that increases the resolution or sampling density of the graph.

Forward heuristic search in belief space

The most conceptually straightforward approach to this problem is to perform a search in *belief space* [7, 67] to find solutions to uncertain robot motion planning problems. The basic idea is that a state of the whole system consists of a robot configuration and a current belief state. In this problem, the belief state is (q, v), with $q \in Q$ and $v \in \mathbb{P}(W)$ representing the current configuration and the region of the workspace that has been observed by the robot to be collision-free.

The belief state is also completely characterized by the path taken to a given configuration, which allows us to express this as a path-dependent search.

Algorithm 2

procedure VAMP_BEL((Q, E), q_0, Q_{goal})
  $path_0 \leftarrow [q_0]$
  procedure GOALTEST(path)
    ▹ return first path to goal found
    return if $path[\text{end}] \notin Q_{\text{goal}}$
  end procedure
  procedure VALIDTEST($path$)
    return COLLISIONFREETEST($path$) **and** $(\bigcup X(path) = \emptyset)$
  end procedure
  $H(path) \leftarrow \alpha \left| S(\text{MP}((Q, E), path[\text{end}], Q_{\text{goal}})) \setminus v \right|$
  QUEUEORDER($path$) = $D(path) + H(path)$
    ▹ DominationTest described in text.
  return PATHDEPSEARCH((Q, E), $path_0$, VALIDTEST,
      QUEUEORDER, DOMINATIONTEST, GOALTEST)
end procedure

Procedure VAMP_BEL provides an implementation of the belief-space search via a call to PATHDEPSEARCH. The procedure is given, as input,

the graph (Q, E), initial configuration q_0, and goal test Q_{goal}. The set of legal actions A that can be taken from state (q, v) is the set of outgoing edges from configuration q that have the property that their swept volume is contained in the previously-viewed region of the workspace v.

The transition function T moves along the edge to a new configuration and augments v with the region of configuration space visible from the new configuration. In order to drive the search toward a goal state, we define a heuristic which is based on a visibility-unaware path $\textsc{mp}((Q, E), q, Q_{\text{goal}})$ obtained by solving the underlying motion-planning problem to the goal. The size of the swept volume of that path that has not yet been viewed is used as a measure of the difficulty of the remaining problem; α is a constant that makes the units match between a measure on workspace (e.g. m^2) and distance in configuration space (e.g. m). The principle behind using such a heuristic is that a small violation region is easier to cover (less search effort and short path) than a big one. This is a heuristic, and not necessarily true, since a large contiguous region may be easy to cover than a disjoint region, scattered about, with small total area.

To approximate the remaining search effort, we could use 1. configuration space distance as the crow flies 2. configuration space distance on the graph of configurations 3. previous, and also avoiding edges that collide with obstacles 4. previous, and also obeying V-Safety Constraint (akin to solving the original problem) In our first results for forward search, we use method 3 plus a measure of the violation region.

Note that, in this search, it is possible for an optimal path to visit the same configuration more than once (with different visibility states v). Nonetheless, the search space is finite given finite Q, because only finitely many possible visibility states can be reached (at most one for each *set* of configurations in Q).

Theorem 4.2.1 *Algorithm* Vamp_Bel *is correct and complete with respect to configuration space graph* (Q, E).

Proof. It is correct, because if it returns a path, that path is a feasible path to a goal state. The A function only allows the robot to move through space that has already been made visible along the path, so the steps are all feasible, and A^* ensures that the final configuration satisfies the goal test. It is complete, because the search space is finite, no feasible actions are ever disallowed, and A^* is complete. □ □

This algorithm is computationally very complex even on modest graphs because the search must generally consider multiple distinct paths that reach a given robot configuration. The search can be pruned by using a *domination criterion*: state (q, v_1) dominates (q, v_2) if $v_1 \subseteq v_2$, which means that if the search visits a state that is dominated by a state that has already been expanded, it can discard the dominated state. In our experiments, this condition did not

occur frequently enough to be useful; different paths will see slightly different regions. On the other hand, in the setting of Bry and Roy [7], the belief space is lower dimensional (covariance matrices of the dynamical state space) and so domination happens much more frequently and makes the search tractable.

We implemented this algorithm with a very computationally cheap domination criterion that eliminates paths that revisit configurations without having visited any new configurations since the last visit (this eliminates looping paths, among others). For HallwayEasy, Figure 1.3, the heuristic is very effective at guiding the search—a solution is found in under 10 sec after expanding 500 search nodes. However, on HallwayHard, Figure 1.4, no solution was found after expanding 440K nodes, with 2 million nodes on the queue, with a computation time of over 2 h.

Optimal Visibility-Based Pursuit Evasion [6] provides multiple pruning rules that are relevant to that problem. Some rules are quick to compute, and others provide more effective pruning (without pruning any paths that may be optimal).

We also experimented with a more expensive domination criterion, and used an admissible heuristic of the distance to the goal, avoiding obstacles. A search node (q_1, v_1) dominates (q_2, v_2) if and only if $q_1 = q_2$, and the path to q_1 is shorter than the path to q_2, and $v_1 \supseteq v_2$. This criterion subsumes the cheap criterion mentioned above. This is implemented by maintaining a list of vs for each q, and iterating the list checking the criterion. Furthermore, for each configuration q we maintain the union of all vs that were achieved at that configuration. This is a *least upper bound* on all the vs. If a search node achieves any visibility outside the least upper bound, then we can determine that is not dominated by any previous node without iterating through the entire list. Finally, we iterate the list by visiting the most recent elements first, since these are more likely to dominate the tentative search node. These two enhancements have no effect on the domination decision, but enhance performance substantially. However, this method is still only practical for very tiny domains.

Reverse search

One may wonder whether a formulation exists to do search backwards from the goal to, for example, do bi-directional search. We sketch how to perform a search to solve VAMP that operates in reverse, growing a search tree rooted at the goal configuration toward the initial configuration. The backwards dynamics are defined over a space denoted (q, x), representing the configuration and "borrowed visibility" respectively. Consider a backwards transition

from $(q_{i+1}, x_{i+1}$ to $(q_i, x_i$. x_i is given by

$$x_i = (x_{i+1} \cup S(q_i, q_{i+1})) \setminus V(q_i)\,. \tag{4.9}$$

Consider VAMP problems with exactly one goal configuration, and no goal visibility. The initial state in the backwards problem is $(q_{\text{goal}}, \emptyset)$. The goal state is $(q_{\text{init}}, \emptyset)$.

Searches with aggressive pruning

The approaches in this section are not complete in general, but may be complete for some robots (e.g. with ball visibility, see Section 2.3); they will prove useful as a subroutine in later algorithms.

In published work [68], we refer to "local-visibility search", since these search strategies are ineffective for problems where the swept volume at one segment of the path requires visibility acquired much earlier (non-locally) in the path. These algorithms are also "local" in contrast to the algorithm described in Section 4.2 that maintains a single visible region to be shared (globally) across all search nodes.

VAMP_GENERIC_VIS is defined in Algorithm 3. The basic version of the method has the same arguments as VAMP_BEL, but it may also be used in *relaxed* mode, which is signaled by an $mask \subset W$, which is the workspace region where the visibility constraint is enforced. By default, $mask = \mathscr{W}$, which is the *unrelaxed* problem. In any mode, it may optionally be given a heuristic function.

Algorithm 3

```
1:  procedure VAMP_GENERIC_VIS(G, path_start, Q_goal, mask = 𝒲)
2:      procedure VALIDTEST(path)
3:          return COLLISIONFREETEST(path) and
                (⋃ X(path) ∩ mask = ∅)
4:      end procedure
5:      procedure QUEUEORDER(path)
6:                  ▹ The violations are penalized regardless of mask.
7:          return D(path) + ∑_i |X(path)_i| + HEURISTIC(path)
8:      end procedure
9:      procedure DOMINATIONTEST(p1, p2)
10:                         ▹ consider only first path to a configuration
11:         return p1[end] = p2[end]
12:     end procedure
13:     procedure HEURISTIC(path)
14:         return min_{q∈Q_goal} D(q, path[end])
15:     end procedure
16:                                 ▹ GoalTest same as in algorithm 2.
17:     return PATHDEPSEARCH(G, path_start, VALIDTEST,
                QUEUEORDER, DOMINATIONTEST, GOALTEST)
18: end procedure
```

When it is not relaxed, it allows traversal of any edge $(q, q') \in E$ whose swept volume is entirely contained in the visible region of the path leading to q (the constraint is enforced everwhere). The choice

of DominationTest prevents the search from considering multiple paths to a given robot configuration. For some combinations of robot kinematics, visibility, and environment this algorithm will be complete. For example, a robot with a wide field of view will always be able to see the space it is about to move into, and so the path taken to a configuration does not affect the feasibility of future segments. However, this method does not suffice for situations in which a robot must move into space that is not immediately visible to it. Whatever visibility we have the first time we expand a configuration q is the visibility that will be permanently associated with it. The algorithm is incomplete because it might commit to a path to some configuration that is not the best in terms of visibility, and it cannot contemplate paths that must revisit the same configuration.

Relaxed mode is used to compute intermediate subgoals, and the robot is allowed to move into areas of the workspace that have not yet been seen, but these motions incur an extra cost. It is not, however, still required to satisfy the constraint in the region specified by $mask$, a feature which is used in the Tourist algorithm of Section 4.2. Relaxed mode is "complete", in the sense that the planner will return a path if one exists under the relaxed constraints. Ideally, the relaxed planner would solve a *minimum-swept volume* problem, keeping track of regions that have ever been violated, and not double-counting the regions that experience repeat violations. As we describe in Section 3.9, this itself is a path-dependent problem and exact solutions are not amenable to efficient search. To keep the computational tractability of sub-problem, so we simply penalize additively the area traversed through unseen regions.

It is possible that this method produces feasible paths, in the same way that the relaxation of an integer linear program may "accidentally" have an integer solution.

In some cases, it is possible to minimize this penalty by an elaborate, zig-zagging path. This occurs when α is too high, and it produces a scattered region of visibility violations. The region, though small in area, is disjoint, and so a path that achieves visibility of this region is itself elaborate. This pathological behavior is possible due to the heuristic nature of the algorithm.

Tree-visibility tour

In this section we present an algorithm that creates a tree that resembles a search tree, but does not have the semantics of a search tree. We are led to this algorithm by the observation that if P is a path that satisfies Equation V-Safety Constraint, then so does $P;\text{reverse}(P)$.

This is guaranteed by the following condition placed on the swept volume.

$$S(q_i, q_j) = S(q_j, q_i) \forall q_1, q_2 \tag{4.10}$$

Furthermore, visibility is monotonic, that is, as the robot moves through the world, after any discrete path step and observation, the visible region is non-decreasing.

$$V([q_1, \ldots, q_{n-1}]) \subseteq V([q_1, \ldots, q_n]) \tag{4.11}$$

These observations lead us to an algorithm that is complete and much more efficient at finding solution paths for VAMP problems than belief-space search, although we will find that it will generally be unsuitable in practice. The purpose of the algorithm is instead to emphasize how to leverage the structure of assumptions 4.10 and 4.11 in VAMP problems.

Rather than associating a new visibility region v with each state in the search, we will maintain a single, global $v \in \mathbb{P}(W)$ and carry out a search directly in Q. The search can only traverse an edge if its swept volume is contained *in the workspace that has been viewed during the entire search process up until that time*. Once this process reaches a goal state, the tree, in the order it was constructed, is used to construct a solution path. Pseudocode is shown in Algorithm 4.

Our first implementation of this concept was based on Rapidly-exploring Random Trees [1], inheriting the probabilistic completeness property. It is also worth pointing out a connection to relaxations in classical search. In a classical search (automated planning) formulation of this problem, there is an atom for each configuration that the robot can attain. This search strategy is analogous to a *delete relaxation* on those fluents, allowing the robot to be at multiple configurations simultaneously.

It proceeds in two phases. First, it constructs a search tree, where the extension from a point in the tree is made only within the region that has been visible from any configuration previously visited in the search. Second, it constructs a path that visits all of the configurations, in the order in which they were added to the tree, and returns that path. The tree search is slightly unusual, because which edges in the graph can be traversed depends globally on all search nodes in the tree. For this reason, we perform a queue-based search, keeping an agenda of *edges*, rather than nodes. If an edge is selected for expansion, but is not yet traversable, it is added back to the end of the agenda for reconsideration after some more of the tree has been grown. When a goal state has been reached, we extract a path from the tree. This path will visit the configurations in the same order that they were visited by the search, but they must be connected together via paths in the tree that existed at the point in the search when the configuration was visited.

Theorem 4.2.2 *VAMP_TREE is correct and complete with respect to the configuration-space graph* (Q, E) *for any robot such that Equation* 4.10

holds.

Proof. It is correct because, if it returns a path, it is a feasible path to a goal state. The set of edges (q_1, q_2) added to p on iteration i of the path-construction phase have the property that either (q_1, q_2) or (q_2, q_1) is in $T[0 : i - 1]$, which, by construction of the tree T and the reversibility assumption in the theorem statement, means that $S(q_1, q_2) \subseteq V(\text{visited}[0..i - 1])$. This, in turn, implies that the path is feasible. The last configuration in visited clearly satisfies the goal test, and it is also the last configuration in the returned path p.

To show that it is complete, we must show that if a feasible path to a goal state exists in (Q, E), the search will find it (or another feasible path). Assume $[q_0, \dots, q_n]$ where $Q_{\text{goal}}(q_n) = \textit{true}$ is a feasible path and assume, for the sake of contradiction, that the first **while** loop cannot add all of the configurations $[q_0, \dots, q_n]$ to *visited*. Then there must be a point in that loop when $[q_0, \dots, q_i]$ are in *visited* for some $0 \leq i < n$, but the algorithm cannot reach q_{i+1}. We know by the assumption that this is a feasible path, so $(q_i, q_{i+1}) \in E$ and $S(q_i, q_{i+1}) \subseteq V([q_0, \dots, q_i]) \cup v_0$, which means q_{i+1} must be in $A(q_i)$. We also know that (q_i, q_{i+1}) will be in *agenda*, because q_i is in *visited*, so it was added, but q_{i+1} is not in visited, so that edge has not been removed from the agenda. But if (q_i, q_{i+1}) is in the agenda and q_{i+1} is in $A(q_i)$, then q_{i+1} can be added to *visited*, and so we reach a contradiction. Thus, we have shown that after the **while** loop, q_n has been reached, and so the algorithm continues to the second phase. The only possible failure mode of the second phase is if *shortest_undirected_path* fails; but, by construction, both $q_{\text{curr}} = \text{visited}[i - 1]$ and $q_{\text{next}} = \text{visited}[i]$ are in $T[0 : i]$, as are paths from q_0 to each of them. Thus we know that there is, at worst, a path going from q_{curr} up to q_0 and back down to q_{next}, by the reversibility assumption. So this loop will terminate and a path p will be returned. □ □

Visibility preimage backchaining

Our final approach to this problem is to perform a much more goal-driven search to observe parts of the workspace that will make desired paths feasible. This algorithm is motivated by the observation that goals can be decomposed into subgoals, as demonstrated in Figure 1.5. Furthermore, VAMP as we have defined it has a strong property of monotonicity (Equation 4.11). Paired with the *reversibility* condition (Equation 4.10), this means that motions in the configuration space are undoable, in a manner that preserves the information

The concept of goal regression and backwards reasoning is traditional in AI and robotics [69].

space. These properties allow us to formulate an algorithm that commits to some parts of a solution without worrying about dead-ends. With this motivation in mind, we describe a general algorithm that has several special cases of interest, described in Section 4.5.

We make use of the Tourist algorithm, whose goal is to see some part of a given previously-unobserved region of workspace. It uses a local-visibility algorithm to find a path, but where the goal test for a configuration is that it is possible to observe some previously unobserved part of the workspace from there.

Heuristic for exploration

A critical aspect to making this search effective is to use a heuristic that drives progress toward the objective of observing part of a region of interest, R. We begin by computing a scalar field, F, in workspace, of the shortest distance from location x to a point in R. Then, the heuristic is $H(q) = \min_{x \in V(q)} F(x)$, which assigns 0 heuristic value to a configuration that can see part of R (because it will be able to see a workspace point x with $F(x) = 0$) and increasingly higher heuristic values to configurations that can only see points that are "far" in the sense of F from R. Computing F is relatively inexpensive, and it effectively models the fact that visibility does not go through walls. This heuristic is illustrated in Section 4.2: the black nodes are the workspace target region R. The figure illustrates the level sets of F.

Algorithm 4

procedure VAMP_TREE((Q, E), V, q_0, Q_{goal}, v_0)
    agenda $\leftarrow [(q_0, q') \text{ for } (q_0, q') \in E]$
    visited $\leftarrow [q_0]$; $T \leftarrow [\ ]$; $v \leftarrow v_0$
    while agenda is not empty **do**
        $(q_s, q_e) \leftarrow$ pop(agenda)
        if $q_e \in$ visited **then continue**
        end if
        if $S(q_s, q_e) \subseteq v$ **then**
            visited.append(q_e) ▹ add conf to path
            T.append((q_s, q_e)) ▹ add edge to tree
            if $Q_{\text{goal}}(q_e)$ **then break**
            end if
            $v \leftarrow v \cup V(q_e)$ ▹ add new visibility
            agenda.extend($[(q_e, q') \text{ for } (q_e, q') \in E]$) ▹ add outgoing edges to agenda
        else
            agenda.append((q_s, q_e)) ▹ save edge for reconsideration
        end if
    end while
    if not $Q_{\text{goal}}(q_e)$ **then return** Failed
    end if
    $p \leftarrow [q_0]$; $q_{\text{curr}} \leftarrow q_0$
    for $i \in [1..\text{len(visited)}]$ **do**
        $q_{\text{next}} \leftarrow$ visited[i] ▹ link configurations using previously-enabled edges
        p.extend(shortest_undirected_path(q_{curr}, q_{next}, $T[0:i]$)[1 :])
        $q_{\text{curr}} \leftarrow q_{\text{next}}$
    end for
    return p
end procedure

This choice of heuristic causes some plateaus, and there may be even moments when the heuristic value increases while progress is made toward a goal configuration. Section 4.2 explores some of these issues.

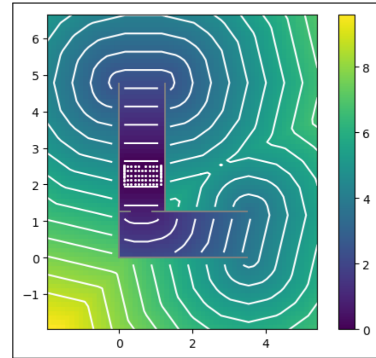


Figure 4.2: Level sets and heatmap of field F used to compute heuristic for acquiring visibility of the white points in the region surrounding $(1,2)$.

Algorithm 5

1: **procedure** TOURIST(G, $path_{start}$, $GoalRegion$, $mask = \mathcal{W}$)
2: ▷ ValidTest, QUEUEORDER, DOMINATIONTEST from algorithm 3.
3: **procedure** GOALTEST(path)
4: ▷ see any of the goal region
5: **return** $GoalRegion \cap V(path) = \emptyset$
6: **end procedure**
7: $H(q) = \min_{x \in V(q)} F(x)$ ▷ where F is distance field
8: **procedure** HEURISTIC($path$)
9: **return** $H(path[\text{end}])$
10: **end procedure**
11: **return** PATHDEPSEARCH(G, $path_{start}$, VALIDTEST, QUEUEORDER, DOMINATIONTEST, GOALTEST)
12: **end procedure**

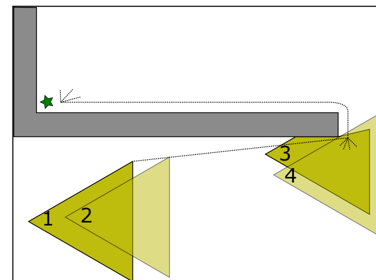


Figure 4.3: The target region to see is marked by a star. Dashed lines indicate the shortest path to the target from a few points in the workspace. Visible regions are labeled 1-4. A motion from 1 to 2 decreases the heuristic value. A motion from 3 to 4 decreases the heuristic value, even though it might be going away from the target region. From 4, there are no incremental translations to further decrease the heuristic value.

Now we can describe the VAMP_BACKCHAIN algorithm, with pseudocode shown in Algorithm 7. The main loop of the algorithm is in lines 3–16. It keeps track of *path*, the solution path it is constructing. On every iteration, it checks to see whether a goal state is reachable from *path*[end] with the visibility $V(path)$. If so, it appends the path that does so to p and returns a final solution. If that test fails, then it generates a path that is guaranteed to increase the visible region (if the problem is feasible), ideally in a way that makes it easier to reach a goal configuration. In line 8, we find a *relaxed plan* p_{relaxed} that reaches a goal state, preferring to stay inside $V(path)$, but allowing excursions if necessary. Now, our objective is to find a path p_{vis} that will observe some previously-unobserved part of the swept volume of p_{relaxed}, by calling procedure TOURISTBOOST. If that call fails, then we fall back on an undirected exploration strategy, to view any part of the unseen workspace. Once we have found a view path, we test to see if we can now find a path to the goal, etc.

Figure 4.4 illustrates the operation of VAMP_BACKCHAIN in the TWOHALLWAY domain. It is given a goal to see the region at the end of the vertical hallway (red points in Figure 4.4a). The hallway is keyed, and the robot can only see the region through the peephole. The planner generates a relaxed plan (gray) to see these points. This relaxed path is in violation, and requires regions be made visible (blue points in Figure 4.4b) before it can be executed. The planner recurses one level, with the blue region as the goal, and both marked "out of bounds" via $mask$, and generates the gray path in

Algorithm 6

1: **procedure** TouristBoost($G, path, GoalRegion, mask = \mathscr{W}$)
2: $\quad p_{\text{vis}} \leftarrow$ Tourist($G, path, GoalRegion$)
3: $\quad$ **if** $p_{\text{vis}} \neq$ **null then**
4: $\quad\quad$ **return** p_{vis}
5: $\quad$ **end if**
6: $\quad mask' \leftarrow mask \cup GoalRegion$
7: $\quad p_{\text{relaxed}} \leftarrow$ Tourist($G, path, GoalRegion, mask'$)
8: $\quad$ **if** $p_{\text{relaxed}} \neq$ **null then**
9: $\quad\quad GoalRegion' \leftarrow \bigcup X(p_{\text{relaxed}})$
10: $\quad\quad$ **return** TouristBoost($G, path, GoalRegion', mask'$)
11: $\quad$ **end if**
12: $\quad$ **return null**
13: **end procedure**

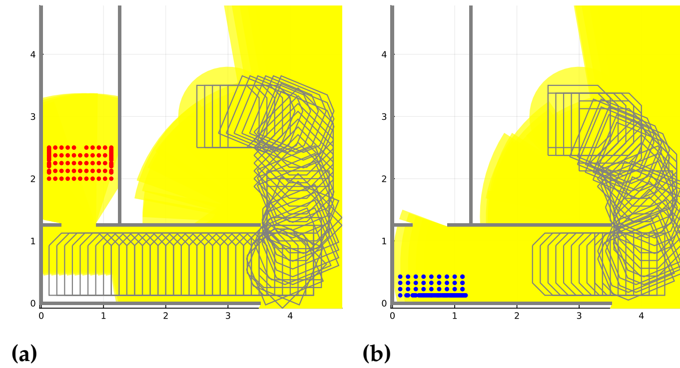


Figure 4.4: Two levels of relaxed planning with a visibility goal.

Figure 4.4b. This path satisfies the full constraints, and it is returned by the planner. Note that the returned path does not satisfy the original goal, but achieves visibility to enable solving the original visibily goal in a later call to the planner.

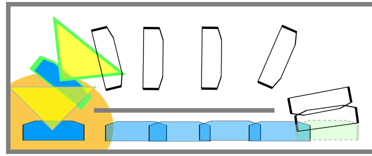
Two difficult examples that motivate the structure of the Vamp_-Backchain algorithm are illustrated in Figure 4.5.

In Figure 4.5a, the robot must move to the dashed outline on the right. It cannot do so with step-wise visibility (line 4), so it makes a relaxed plan (line 8) to slide horizontally to the goal. However, none of the swept volume of that relaxed plan can be viewed (line 2) under normal visibility constraints, nor can we even generate a relaxed plan to view it (line 7). We fall back on simply generating a path that views some part of the unseen workspace (line 12) which yields the path shown by the unfilled robot outlines. The ultimate solution to the problem is indicated by the robot outlines.

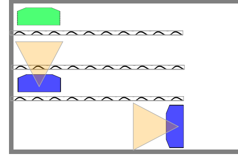
In Figure 4.5b, we see an example illustrating the potential need

Algorithm 7

```
1: procedure VAMP_BACKCHAIN(G, q_start, Q_goal)
2:     path ← [q_start]
3:     while true do
4:         p ← VAMP_GENERIC_VIS(G, path, Q_goal)
5:         if p ≠ null then
6:             return p
7:         end if
8:         p ← VAMP_GENERIC_VIS(G, path, Q_goal, mask = ∅)
9:         GoalRegion ← X(p)[end]
10:        p ← TOURISTBOOST(G, path, GoalRegion)
11:        if p = null then
12:            path ← TOURIST(G, path, GoalRegion =
               𝒲 \ V(path))
13:        else
14:            path ← p
15:        end if
16:    end while
17: end procedure
```



(a)



(b)

Figure 4.5: Difficult examples for the VAMP_BACKCHAIN algorithm.

for arbitrary recursive nesting. In this case, the inner walls are transparent (so the robot can see through them, but it cannot move through them.) The solution requires moving forward into the bottom-most hallway to clear it, then moving into it again sideways to look through the windows, thus clearing the hallway above it, and so on.

Theorem 4.2.3 *The algorithm VAMP_BACKCHAIN is correct and complete with respect to the configuration-space graph* (Q, E) *for any robot such that Equation* 4.10 *holds.*

Proof. If the algorithm returns an answer, it is a feasible path to a goal state. The path is feasible because it is a concatenation of paths made by non-relaxed calls to VAMP_PATH_VIS (either directly or via calls to TOURIST), and those paths are feasible by construction. The final call to VAMP_PATH_VIS guarantees that the final configuration satisfies Q_{goal}.

To show that it is complete, we begin with some lemmas.

Lemma 1. The Tourist procedure is guaranteed to terminate, and either return a path that will visit a configuration that has not been reached before or fail.

Lemma 2. If there is a feasible path and the call in line 4 and line 8 fails, and the call to TouristBoost fails, then the call to Tourist in line 12 is guaranteed to return a path that will visit a configuration that has not been visited before.

By Lemmas 1 and 2, on every iteration of the main **while** loop, either the call to Vamp_Path_Vis succeeds and finds and returns a solution path, or, a sequence of configurations will be added to the path that causes some not-yet-viewed space to be seen.

The while loop terminates. If a path does not exist, then eventually all the space that can be seen will have been seen and the call on line 12 will fail, and the algorithm will terminate with failure. If a solution path $[q_0, \ldots, q_n]$ does exist, then because repeated calls on line 12 will eventually visit all configurations that can be reached on feasible paths, and therefore see all the space that can be seen, then there is a point at which all of $S([q_0, \ldots, q_n])$ will be in v and so a call to Vamp_Path_Vis on line 4 will return with a solution. □ □

In this formulation of Tourist, the search is over as soon as any of the goal region is seen. The entire region can be seen, if possible, by repeating the search with the remaining region. In published work [68], we used an alternative formulation in which the search attempts to see "as much as possible" of the goal region. In essence, this formulation generalizes a goal test. It also separates the notion of the order in the priority queue from the quality of the path. The search is not allowed to terminate until some of the goal region is seen, but once some of the goal region is seen, the search terminates only if there has not been recent progress. There is a clear notion of progress based on the size of the region that remains to be seen, but it is challenging to determine what counts as "recent" in a principled manner. This strategy may be interesting for practical reasons, but we choose to describe the straightforward formulation here.

4.3 Reformulation

Here we provide a formulation that sheds some light on the algorithms above.

Pruning

Instead of searching over all possible configuration-space paths, we search over some subset. Moreover, this subset may be dynamically determined. A projection function maps a path to a *hash* space. In almost all of our experiments we consider specifically the projection function that looks at the final configuration of the path:

$$\pi(P) = P[\text{end}] \tag{4.12}$$

In this case, $\pi(P) \in \mathscr{C}$, but other choices are possible. In the extreme case, the projection function corresponds to computing the *path summary*, and leads to a search over all relevant paths (which seems generally intractable).

Additionally, for a given hash, the domination criterion determines which paths to keep. In our experiments we consider specifically the criterion which keeps the first (according to the search order) path for a given hash. In this case, each hash value is associated with up to one path, but other choices are possible.

Constraint relaxation

Because of the pruning, there are usually no paths that solve the problem. The easiest witness for this assertion is to consider a VAMP instance for which any solution path must contain the same configuration twice. With the projection function and domination criterion given as examples above, the search will never consider this path.

To get useful information from running this search, the visibility safety constraint (Equation V-Safety Constraint) is relaxed into a cost penalty. The paths produced by this search are not (at least not on their own) solutions, and intuitively the penalty should reflect how much "effort" is required (by P_{pre} below) to restore the solution.

There are multiple possible formulations for the penalty. The approach in Section 4.2 paper penalizes a path P by (possibly) over-counting the violation set:

$$\sum_{i=1}^{\text{length}(P)} \mu(X(P)[i])\,. \tag{4.13}$$

A path that sweeps a given region of $\mathscr{W}$ multiple times without acquiring additional visibility is penalized more than a path that sweeps the given region just once. This approach was chosen at that time because, relative to that implementation, it would require less computation because we must only keep track of a scalar per candidate path.

The more principled approach avoids over-counting:

$$\mu\left(\bigcup_{i=1}^{\text{length}(P)} X(P)[i]\right)\,. \tag{4.14}$$

This choice requires either keeping a representation of the iterates (sets in $\mathscr{W}$), or computing the penalty iterating through the path. Both of these are less efficient in memory and/or time than over-counting, though note that computing $X(P)[i]$ requires computing $V(P)[i-1]$,

and so an implementation already needs either a representation of that set, or to compute the set by iterating through the path.

To represent the relaxed problem in the same framework, we augment the formulation of a VAMP problem instance with the following fields:

Safety Violation Penalty	$c : (\mathcal{C})^* \rightarrow \mathbb{R}$
Safety Constraint Mask	mask $\subseteq \mathcal{W}$

and redefine Equation V-Safety Constraint as follows:

$$X(P)[i] \cap \text{mask} = \emptyset \,, i = 1 \cdots n \,. \quad \text{(Masked V-Safety Constraint)}$$

By default, mask = $\mathcal{W}$. In the relaxed setting mask = $\emptyset$.

Goal regression

The relaxed search above generates two results: 1. a path P that is generally not a solution to the original VAMP instance D_{init}. 2. a region $v_{\text{missing}} \subseteq \mathcal{W}$:

$$v_{\text{missing}} = \bigcup_{i=1}^{\text{length}(P)} X(P)[i] \,. \tag{4.15}$$

If v_{missing} were somehow made visible initially, then we would have a solution to the original instance. In other words, let D_{solved} be a VAMP instance like D_{init}, except that

$$D_{solved}.v_{\text{init}} = D_{\text{init}} \cup v_{\text{missing}} \tag{4.16}$$

P is a solution to D_{solved}.

The problem at hand now is to compute a path P_{pre} for a domain D_{pre} with $D_{\text{pre}} Q_{\text{goal}} = \mathcal{C}$ and $V_{\text{goal}} = \{v_{\text{missing}}\}$ ("see-all") or $V_{\text{goal}} = \{v | v \subseteq v_{\text{missing}}\}$ ("see-any").

A see-any problem is a relaxation of a see-all problem where a conjunction is replaced by disjunction in the goal condition. This relaxation is easier to solve. We can solve the see-all problem by repeatedly solving see-any in a receding horizon fashion.

This path acquires visibility of v_{missing}. (If we choose V_{goal} corresponding to "see-all", and we further constrain $Q_{\text{goal}} = \{D_{\text{init}}.q_{\text{init}}\}$, then we could simply concatenate P_{pre} and P to get a solution. This approach would generally produce a longer path at the benefit of not needing to solve a new planning instance with $q_{\text{init}} = P_{\text{pre}}[\text{end}]$).)

In general, computing P_{pre} may itself be an intractable problem. We apply the same pruning as above and relax the constraint into a

penalty, to compute P'. Critically, we re-introduce a constraint on $X(P')$ to ensure that a solution to this subproblem makes progress toward solving the overall problem. mask = v_{missing}

Just as before, P' is generally not a solution to D_{pre} (with mask = $\mathcal{W}$), and induces an analogous v_{missing}. Goal regression continues and relaxed searching continues until the relaxed search generates a path that happens to solve mask = $\mathcal{W}$. If $\mathcal{W}$ is bounded, then this is guaranteed.

4.4 Complexity

Complexity of problem class

VAMP is a generalization of classical motion planning, which is PSPACE-hard [70]. There are, however, restricted classes of classical motion planning that are in P. For example, there are algorithms that require time exponential in the dimension of the configuration space, but are polynomial in all other quantities (such as the number of obstacles and the degree of polynomial defining the obstacles). If we consider the class of classical motion planning problems with bounded dimensionality of the configuration space, that class is in P. However, when we demand shortest paths, the problem is NP-hard even for finding the shortest path for a point in a 3D workspace [71].

There are generalizations of motion planning problem ([72, 73]) that are computationally hard even when considering this restricted class among feasibility. We conjecture that finding optimal VAMP paths is NP-hard in planar domains in an (x, y, θ) configuration space. The argument presented for an abstracted version of VAMP in Section 3.8 shows that this abstracted version is NP-COMPLETE, but the vertex visit graphs arising from VAMP instances is more restricted than arbitrary vertex visit graphs.

Complexity of solution length

Section 3.8 demonstrates the existence of an abstracted path search problem, without any geometry, exhibiting quadratic scaling of the length of a solution with respect to the size of the problem. We can construct VAMP problems with this property, too, in Figure 4.6. The robot must start in the green configuration at the lower left and navigate to the red configuration in the top right. An example configuration is shown in purple with the corresponding view region.

The view region does not allow the robot to translate sideways (strafe) locally. Only the bottom part of the domain contains enough room for the robot to change orientation. The length of a solution grows quadratically with the number of steps in the domain (here a domain with 5 steps is shown), since the robot must return to the bottom each time it makes a step's worth of progress toward the goal.

Compared to some other motion planning problems that remain computationally hard even when restricted to a low-dimensional configuration space, such as minimum constraint removal and motion planning with obstacle uncertainty [72, 73], VAMP stands out by having solutions whose lengths scale quadratically in the worst case. There are instances of Visibility-based pursuit evasion that require shadows to be recontaminated linearly many times, but this does not necessarily result in paths with quadratic length. An example is available in Theorem 7 and Figure 3 of Guibas et al. [74], and the same example is reproduced in Figure 4.7.

Complexity of algorithm

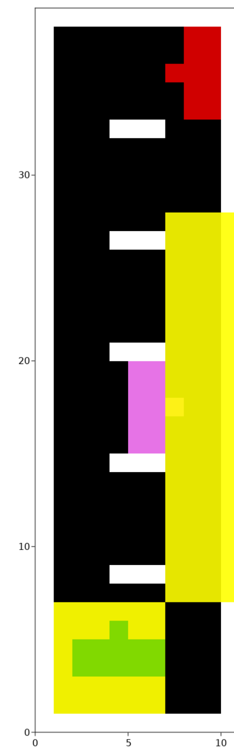


Figure 4.6: Domain illustrating worst-case (quadratic) scaling of solution length with domain size.

We provide here some quantities that can be used in analyzing the algorithmic complexity of the procedures mentioned in this chapter. For a configuration graph with c vertices and a workspace with w cells, there are at most 2^c distinct visible regions. This is still huge, and also may be considerably less than 2^w, the number of "occupancy grid" states of the discretized workspace. Both of these numbers may wildly overestimate the number of *reachable* distinct visible regions, since there are many distinct sets of configurations that induce the same visible region, even more so when there are occluders.

One difficulty in stating the algorithmic time complexity is determining the time complexity of geometric operations. Chapter 5 enables a trade-off between time and memory for determining the incremental violation region $X(p)$, and contains a discussion of the geometric operations. We leave a careful analysis of the complexity of the algorithms for future work.

VAMP instances are challenging when the domain requires plans that achieve visibility in order to perform a motion in the future. We present two small instances, HallwayHard and TwoHallway that have this property. Solving the problem directly in the belief space is computationally intractable. We, instead, direct the search by relying on calls to constraint-relaxed plans.

4.5 Experiments and planning results

In our experiments, we consider a planar robot (1 m × 1 m) operating in a 2D workspace. For all of the experiments, we discretize the robot motions and search on a 6-connected lattice ($\Delta x = \Delta y = 0.125$ m, $\Delta\theta = \frac{2\pi}{16}$). The depth of view of the visible region is 2.5 m. All swept volume and containment computations were performed by sampling points along the boundary of the robot.

We ran two versions of VAMP_BACKCHAIN. VB∞ corresponds to the algorithm as presented in Section 4.2, with the difference that all sub-calls to the planners are relaxed versions. We never call the unrelaxed planner, however we still verify the feasibility of paths before incorporating them into the final path. This choice has the benefit of accelerating the search procedure, since we do not have to wait for TOURIST to return failure in situations where the search is incomplete. In practice, the relaxed planners often return a feasible path if one exists, but occasionally they produce a violating path, which means subsequent searches may do unnecessary work to provide visibility in the violated region.

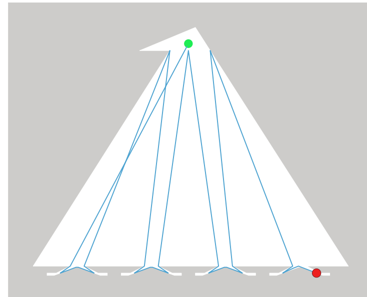


Figure 4.7: Domain illustrating back-and-forth solution path in visibility-based pursuit evasion. The path starts at the top green marker, and the goal is to ensure all regions are uncontaminated. The optimal solution takes the path in blue, ending at the red marker, and must clear the nook at the top repeatedly. The length of the path depends linearly on the number of hooks on the bottom, and linearly on the height of the domain. This figure is based on Figure 25(d) in Stiffler et al. [6].

For example, for a robot with a wide view cone approaches a wall following $q_1, q_2, \cdots$, we have $V(q_1) \supset V(q_2) \supset \cdots$.

VB1 corresponds to VAMP_BACKCHAIN, but with a recursion depth limit. In this variation the call to TOURISTBOOST is skipped. Furthermore, instead of waiting for the call to TOURIST to fail we set a timeout, to trigger the subsequent call to TOURIST.

We run experiments on many instances of VAMP problems. Instances vary in obstacle, start and goal states, and field of view of the vision sensor.

There are three combinations of obstacle layout and start/goal states, each exhibiting increasing problem difficulty: HALLWAYEASY depicted in Figure 1.3, HALLWAYHARD depicted in Figure 1.4, and TWOHALLWAY depicted in Figure 1.5, which contains a "keyed" vertical hallway, which can only be entered backwards.

For each experiment, we report search time, path length, and total number of nodes expanded in any subroutine searches.

TWOHALLWAY is designed to demonstrate the recursion capabilities of
VAMP_BACKCHAIN, so VB∞ noticeably outperforms VB1 on it. Because VB1 does not perform backchaining more than once, it relies on line 12 of VAMP_BACKCHAIN. In practice, for problems exhibiting the nested dependency as in TWOHALLWAY, VB1 generates paths that view the whole space because the search cannot be guided through nested dependencies.

Table 4.1: Planning results across varying viewcones

		fov=50°		fov=200°		fov=350°	
		VB1	VB∞	VB1	VB∞	VB1	VB∞
Search time (s)	HallwayEasy	5.1	16.4	1.2	1.2	1.0	2.2
	HallwayHard	23.4	315.3	9.4	13.9	2.4	3.5
	TwoHallway	2868.3	281.3	638.5	220.9	1952.1	134.8
Path length (m)	HallwayEasy	12.3	13.3	8.4	8.4	8.4	8.4
	HallwayHard	14.3	16.9	12.5	12.5	11.4	11.4
	TwoHallway	63.9	43.4	47.6	43.2	40.6	34.3
Closed nodes	HallwayEasy	2578	9241	377	377	137	137
	HallwayHard	7667	40428	3436	4469	604	604
	TwoHallway	139484	64145	76083	62586	92184	44188

Situations in which VB∞ performs worse are due to suboptimal relaxed paths, which incur violations that could be avoided.

We also collected search times and tree size for the TreeVis algorithm. For TwoHallway, it expands 62,000 nodes and searches for 60 seconds. Note that this does not include any time for generating a path. The naïve path would include every edge in the tree, visiting every node in search order, which would never be a practical path. Note additionally that TreeVis is not a directed search, and so in domains where the workspace is large, it is unlikely that TreeVis will be practical.

4.6 Appendix: Geometry

The language of set theory gives us a compact way to express the above formulation. However, to compute with these sets, we must choose a representation. In this dissertation, we have considered three classes of representations: boundary representations (such as meshes, polygons, and polyhedra), uniform raster grids (pixel/voxels), and sampling points (point clouds).

Representing the sensing region

A camera's data sheet specifies the horizontal and vertical field of view, along with a measurement range. These specifications can be used to construct a view frustum consisting of the intersection of 6 half-spaces. We assume that if an obstacle is in the view frustum, it will be sensed.

Visibility computations

We have considered several methods for performing the geometric computations required, specifically computing the visibility violation region (either its size or its extent). Each method imposes a set of requirements on the representation of the robot and occluder geometry. Note that the logic of performing collision detection is different from computing visibility violations.

Visibility constraint versus collision detection

Given obstacle/occluder regions o_i, s is collision-free iff

$$s \cap (o_1 \cup o_2 \cup \cdots) = \emptyset \tag{4.17}$$

Distributing gives

$$(s \cap o_1) \cup (s \cap o_2) \cup \cdots = \emptyset \tag{4.18}$$

which can be logically decomposed

$$(s \cap o_1 = \emptyset) \wedge (s \cap o_2 = \emptyset) \wedge \cdots \tag{4.19}$$

On the other hand, s is visibility-safe with respect to visible regions v_i iff

$$s \subseteq (v_1 \cup v_2 \cup \cdots) \tag{4.20}$$

Equivalently

$$s \setminus (v_1 \cup v_2 \cup \cdots) = \emptyset \tag{4.21}$$

which can be written as

$$((s \setminus v_1) \setminus v_2) \setminus \cdots = \emptyset \tag{4.22}$$

The logic of collision detection decomposes in such a way that Equation V-Safety Constraint cannot. If we were specifically interested in $s \subseteq v_i$, we could use a raster graphics rendering pipeline to determine visibility. Assign a sentinel color to the object whose visibility you want to check, s, and render the scene without any occluders from the perspective corresponding to v_i. Count the number of pixels with the sentinel color, and re-render the scene with occluders. If the count is the same, then $s \subseteq v_i$. If the count is reduced, then some of s is occluded. Without knowing the specific region of s that

is occluded, it is not possible to use this procedure to answer our query with respect to $\bigcup v_i$.

We could also use a ray casting operation. If s is convex, we can determine its visibility from a vantage point by constructing the convex hull s and the point. If this convex hull "collides" with any occluders, then some of s is occluded. The convex hull must also be contained within the visible region. But because there may not be any single view that covers s, we need to consider unions of views.

Representing convex polyhedra

There are two complementary representations for convex polyhedra: the half space representation and the vertex representation.

The half-space representation is given by the intersection of finitely many half spaces. Each half-space is given by a normal vector, which by convention points out of the shape, and an offset. A point with coordinates represented as a column vector x is inside the region if $Ax \leq b$, where A is a matrix with row vectors corresponding to the half-space normals, and b is a vector of the half-space offsets. The inequality applies element-wise.

The vertex representation is given by the convex hull of finitely many points and the conic hull of finitely many rays. x is inside the region if $x = Vy + Rz$, where V is a matrix with column vectors representing coordinates of the points and R the direction of rays. $\sum y_i = 1$, $y_i \geq 0$ and $z_i \geq 0$.

Neither of these representations are unique. For the half-space representation, any half-space that contains the entire region can be added. For the vertex representation, any point or ray in the convex or conic hull, respectively, can be added. Note that rays are required in the vertex representation, since the half-spaces may not be closed. This is exactly the case when representing the shadow cast by a convex object. Also note the degenerate cases. The intersection of no half spaces represents the entire domain. To represent an empty region, there should be no solution x.

There are algorithms to convert between representations. This is helpful because different operations between convex polytopes are easiest in different representations. For example, taking intersections of convex polyhedra amounts to combining all their half-spaces. As noted, this does not in general result in a minimal representation.

To see how this representation is not minimal in general, consider taking the intersection of two intricate shapes with no overlap. Determining emptiness of the half-space representation is equivalent to deciding the feasibility of a linear program.

Visible region by constructing shadow volumes

Given a vantage point and a polyhedral occluder, the shadow cast by an occluder can be constructed by first determining its *silhouette*. For example, in 2D, we can detect the silhouette points of an occluder by identifying points that are straddled by two edges, one facing toward and the other away from the vantage point. If the occluder is convex, then the half-space representation of the shadow consists of 1. all of the half-spaces of the occluder facing the vantage point 2. half-spaces generated by the silhouette element (point or edge) and vantage point. Figure 4.8 shows the half-spaces involved in the shadow of a rectangular occluder.

In 3D, we detect silhouette edges that are straddled by two faces.

This process is related to *shadow volumes* in computer graphics. If the occluder is nonconvex, then multiple shadow volumes arise from the object, which does not cause a problem in the graphics use case [75].

A point is visible if and only if it is in the view cone and outside at least one shadow. If we discretize the robot into points, then we can use this procedure to answer approximate visibility queries. On the other hand, we can perform ray cast queries from the vantage point to these queries. This amounts to checking line segment intersection against occluder geometry. This approximation of the problem (it is a relaxation, since fewer constraints apply) is easier to solve, and each discretized point can be checked in parallel.

Constructing unseen swept region

Figure 4.8 illustrates a procedure that explicitly constructs the set difference of two convex polyhedra, representing the result as a union of convex polyhedra.

Once the visible region is constructed explicitly, the same procedure can be used to take a set difference with the swept geometry. What remains is the unseen region.

If the workspace is 2D, there are efficient boundary representations for nonconvex polygons, including those with holes. However, it is challenging to handle boolean operations robustly. Even when maintaining only convex objects, robustness problems arise, e.g. when slivers in workspace are generated. Furthermore, these kinds of computational geometry computations are relatively slow to perform within the planner.

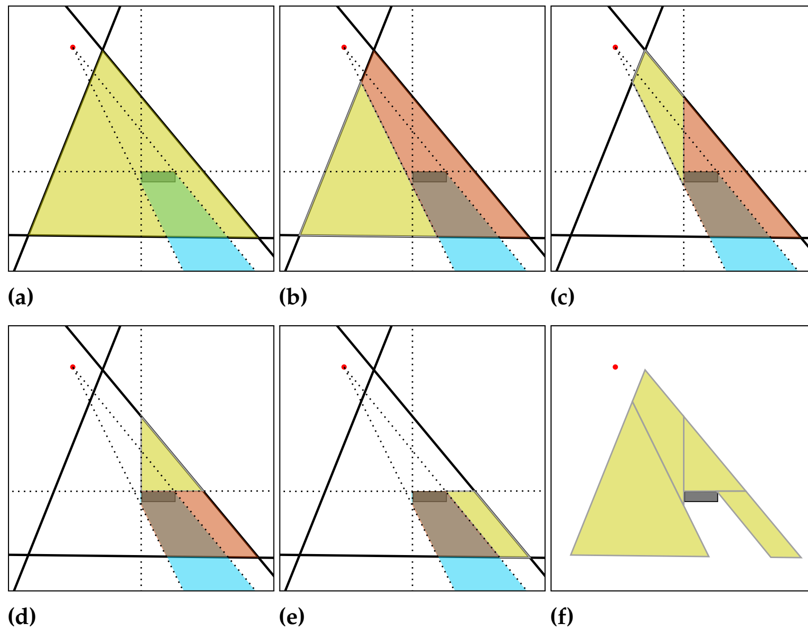


Figure 4.8: Set difference (positive \ negative) for union of convex polyhedra represented by half-spaces. The positive set is the convex yellow region, which is the intersection of 3 half-spaces drawn with a solid boundary. The negative set is the convex blue region, which is the intersection of 4 half-spaces drawn with a dashed boundary. The negative set is the shadow, from the red vantage point, of a rectangle. Figures 4.8b to 4.8e shows each step of the procedure, which splits the yellow region into the two regions, yellow and red, depicted in the subsequent subfigure. Figure 4.8f shows the final result: the union of these four yellow convex regions. Note that the resulting convex decomposition depends on the order in which the half-spaces are considered.

We discussed the approximating the geometry of the robot with point samples. We used this technique in most experiments in continuous-workspace domains. In principle, the solution of this approximate plan can be verified with the approach above. If any violation is found, a point in the unseen swept region can be added to the sample approximation of the robot, and the planning procedure rerun in the modified domain where the approximation has been refined.

Representing the visible region of a path

In the continuous-workspace experiments, we do not explicitly construct the union of all the visible regions. We instead maintain a list of regions. A point is in the union if it is in any of the components. To reduce memory usage and computation time, we attempted an optimization to remove visible regions that do not contribute to the union. This was done approximately, by sampling points in the workspace. A region does not contribute to the union if it does not include any additional points. This solution is undesirable since it is approximate, and requires a choice of sampling procedure (we used a uniform grid).

Discrete workspace

There are many challenges arising from how to represent continuous-space geometry and quantities over said geometry. The way to address most of these challenges is to discretize, so we attempted a formulation of VAMP that is stated completely in terms of a discrete workspace. A set in discrete workspace can be represented computationally as a set of tuples of integer coordinates. Set operations with a geometric interpretation can be performed directly on these sets. The simplicity of this approach has a downside—large sets are expensive to compute on. Hence, in practice, spatial hierarchy is exploited to represent large regions efficiently [76]. We focus on small workspaces to avoid these issues.

For example, we will solve the eikonal equations in the workspace for the heuristic to achieve visibility. As with many partial differential equations, discretization is central to computing solutions.

Collision checking

At first, we desired to stay completely in the world of discrete geometry, sometimes called digital geometry. The occluder and collider geometries, along with the robot swept volume and visible region, were all to be defined discretely. We would have the robot move immediately from one configuration, to the next. To prevent the robot from reorienting itself in a narrow hallway, it would be necessary to make the robot's bounding volume be rectangular, and have a hallway with a width in between the short and long dimension of the rectangle. However, this choice allows the robot to only enter in two orientations, a half-turn apart. The TwoHallway domain requires the robot to enter the hallway in at least three orientations, and so it would not be possible to model this domain in the discrete world.

cf. In video games, the collision region of a sprite does not necessarily match the geometry suggested by the sprite. When collision checking is done in accordance with the rendering of the sprite and colliders, it is called *pixel-perfect* or *per-pixel* collision detection.

We therefore returned to thinking about an underlying continuous workspace geometry and computing a grid supercover to overapproximate the continuous geometry. Because the discretization of the configuration space was chosen to be uniform, we can precompute the discretized swept volume of all motion actions. It would have been sufficient to sample densely along an action transition, but there is a risk of under-approximating the swept volume. We instead used general-purpose off-the-shelf algorithms for computing forward-reachability of a set of initial conditions under linear ordinary differential equations (ODE) [77]. These methods produce over-approximations, and since we aim to over-approximate the continuous set in the discretized workspace anyway, a visual inspection can ensure that the correct volume is computed. More efficient solutions for computing swept volumes certainly exist, but

The concept of a supercover also goes by the name of *conservative rasterization* and *outer Jordan digitization*.

since these sets can be precomputed, we use these general purpose tools.

A linear ODE can be expressed as

$$x\prime(t) = Ax(t)\,, \tag{4.23}$$

where A is a time-varying matrix.

To compute $S(q_1, q_2)$, we set the initial condition set $x(0) \in S(q_1)$, set $A(t)$ according to the motion, and integrate forward for one unit of time (without loss of generality). For example, to rotate a planar set about the origin at velocity r,

$$A = \begin{bmatrix} 0 & -r \\ r & 0 \end{bmatrix} \tag{4.24}$$

To translate, one can use homogeneous coordinates to express an affine equation linearly. Here, we will just state the equation in affine form as

$$x'(t) = Ax(t) + c \tag{4.25}$$

So with $A = 0$, the initial condition set translates by c in unit time. Together, this allows computing the rotation about any point, not just the origin. However, primitive motions are often straight lines in configuration space. For a rigid-body robot with translation and orientation, there is not a single translation or rotation that corresponds to the motion of a straight line in configuration space. The robot rotates about a point that is itself being translated. In other words, there is no static 2D vector field that would cause a set of initial conditions to simultaneously translate, and rotate about a translating point.

It is possible to be more careful about the dimensionality here. c has units of velocity, but multiplied by unit time, it is a displacement. The matrix A is also dimensioned [78].

To use the reachability analysis tools, we express simultaneous translation and rotation by adding dimensions corresponding to the center point of the rotation. We use $\hat{x}$ to represent this lifted space. The motion corresponding to a simultaneous $r/2\pi$ rotation about the point initially at the a, translating by b is given by

$$\hat{x}'(t) = \begin{bmatrix} 0 & -r & 0 & r \\ r & 0 & -r & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \hat{x}(t) + \begin{bmatrix} b_1 \\ b_2 \\ b_1 \\ b_2 \end{bmatrix} \tag{4.26}$$

with

$$\hat{x}(0) = \begin{bmatrix} x_1(0) \\ x_2(0) \\ a_1 \\ a_2 \end{bmatrix} \tag{4.27}$$

To determine the unlifted set, just project onto the original coordinates.

Visibility

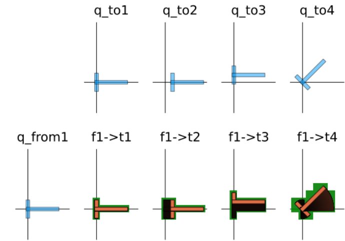


Figure 4.9: Some discretized swept volumes for a planar T-shaped robot.

There is a rich body of work on formulating geometry over uniform grids that has parallels to Euclidean geometry. We attempted to define visibility in terms of digital straight lines, however it produces results that do not correspond to a helpful notion of visibility in continuous geometry. In fact, it reveals a subtle issue with defining visibility, illustrated in Figure 4.10. For cell t to be visible from the center of cell s, the cell below t must be clear. If there is a vertical wall in the first column, including t, then t is not visible. The fix we use is to define a cell to be visible if any of the edges are visible. On the grid, it is sufficient to cast a ray to each corner, and check two adjacent corners, since no occluder is smaller than a grid element.

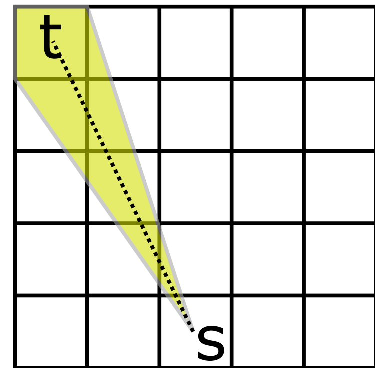


Figure 4.10: For cell t to be visible from the center of cell s, the cell below t must be clear.

Post-processing to minimize views

Each of the algorithms returns a path of consecutive configurations such that, if the robot were to take and process an image at every configuration, the path would be safe. However, when imaging requires the robot to be stationary or the processing is slow, it is desirable to minimize the number of images required while still guaranteeing safety. To select which configurations actually require an image to be acquired and processed, we simply run a greedy set-cover algorithm, and then annotate the configurations in the path to indicate whether the robot should take an image there.

Many mobile-manipulation robots have heads that pan and tilt. If the head is such that moving it substantially changes the robot configuration from a collision-avoidance perspective (e.g., it can extend a periscope) then it may be necessary to include the degrees of freedom of the head in the robot's configuration, q, and apply the algorithms in this thesis directly. However, when moving the head makes a relatively small change in the swept volume of the robot, planning for the head can be decoupled from planning for the rest of the robot. This is an easier planning problem. We do this

in two phases. First, we run one of the VAMP algorithms in the configuration space of the robot, but without including the head's degrees of freedom. We use a visibility function V that includes *the union of all possible views* (the *field of regard*) that can be obtained by moving the head, given the rest of the configuration q. When a path is returned, we post-process by selecting not just what robot configurations require an image but also which orientation(s) the head should have when taking the images. We accomplish this by partitioning the viewable region of space into a finite set and associate a head configuration with each element. Then, when we have a set-cover problem. We run it over the product of the body configurations in the path and the possible head configurations. We used an integer linear programming formulation of minimum set cover. This formulation penalizes the first view at robot configuration more than subsequent views. This can be slow, and a greedy set-cover approach is faster.

4.7 Appendix: Extension to tactile sensing

This material was originally presented in the ICRA 2019 workshop *ViTac: Integrating Vision and Touch for Multimodal and Cross-modal Perception*.

Although it was originally developed for robots with limited visual sensing, the techniques can be generalized to robots equipped with tactile sensors. The formulation we propose allows the robot to make unexpected contact with the environment, so long as contact is guaranteed to happen on specific contact-sensitive surfaces of the robot.

We propose an extended model of safety that is appropriate for robots with both visual and tactile sensors. These models of safety are suitable for motion planning in unknown maps. In the following, we begin by describing our approach to planning with incomplete information [68] and then discuss ways in which it can be directly applied to robots with tactile sensing.

If the robot is entirely covered in tactile sensors [55, 79], then it could proceed without any special planning, assuming it is executing guarded moves (moving slowly enough to not damage itself before it can detect contact and halt). More generally, a robot cannot detect contact as it moves in an arbitrary direction in configuration space. We explore a few cases, starting with a case that reduces directly to the previous formulation of the VAMP problem. To handle more general robots, we require a new formulation of the safety constraint. Finally, we explain the strategies that a planner may exploit when a robot has access to both visual and tactile sensing modalities.

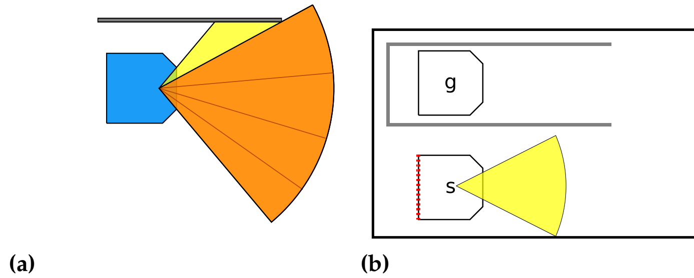


Figure 4.11: (a) Some possible sensed volumes with respect to robot configuration (blue) and environment (gray). The orange region can be covered with an unretractable cane. A few configurations of a cane are shown in black. The yellow region can be covered with a depth sensor (or retractable cane). (b) A robot with vision (yellow) and bump (red) sensing in a domain with start at s and goal at g.

Direct reduction to VAMP domains

We consider a domain that resembles how a blind person sweeps a cane to detect obstacles, in which the previous formulation applies directly. We assume that the cane is permitted to contact the environment and reliably detects contact anywhere along its edge.

If the cane can be retracted radially, as well as be swept angularly, then the problem reduces to the domains where the depth sensor has a limited field of view and depth of view. If the cane cannot be retracted, the problem changes in two ways. 1. the function V behaves differently, since the "occlusions" induced by the known environment obstacles are larger, as shown in figure 4.11a. 2. the cane must be considered in the collision checking, and as such must be added to the planning state space. Even though the configuration of the cane is part of the state space, the planner itself does not need to plan the sweeping action, and does not need to consider every configuration of the cane. We may assume that the cane can be swept within the connected component of configuration space. More specifically, consider an equivalence class such that q_i and q_j are equal if the robot (ignoring the cane) is in the same pose but the cane itself is in different poses, and the cane may move freely from q_i to q_j. The planner may treat q_i to q_j as equivalent in most configurations, except for those that may transition into a configuration in which the configuration space becomes further disconnected.

Extension of safety constraint

The previous section demonstrates a specific robot for which tactile-safe motion planning reduces to a previous VAMP formulation. More generally, the robot may contain contact-sensitive and contact-insensitive surfaces. The planner would deploy the contact-sensitive

surfaces, shielding the contact-insensitive surfaces and leading with the contact-sensitive ones. The tactile-equivalent of constraint V-Safety Constraint depends on the geometry of these surfaces. A motion is valid if the swept volume of the contact-insensitive surfaces does not include any unseen regions of workspace. To implement the check of the visibility-safety constraint V-Safety Constraint, the robot geometry is discretized into points. This approximation enables handling the union without relying on explicit geometry calculations. It is similarly possible to formulate the tactile-safety constraint for point geometries.

Visual and tactile sensing

A combination of vision and tactile sensors allow the option of moving quickly through known-free space and slowly, but leading with a contact sensitive surface in unknown space. To illustrate, consider a robot with a front-facing depth sensor and a rear bumper in figure 4.11b. The goal is still to move about safely in an uncertain environment. The planner decides when to do guarded moves (when it is relying on the tactile modality to guarantee safety), and when to move at full speed (when it is moving through space that has been previously seen or previously traversed). The structure of the VAMP algorithm remains the same, and a primitive motion is only valid if it satisfies equation V-Safety Constraint, or if *throughout* the execution only contact-sensitive surfaces traverse unknown workspace regions. To achieve the goal, the planner for the robot in figure 4.11b may enter the hallway to visually confirm it is clear, then maneuver to the goal (which it cannot do directly, since the robot cannot turn inside of the narrow hallway), or it may execute a guarded move to back into the hallway, leading with the robot's contact-sensitive rear bumper.

Temporal constraints and objectives

It may be beneficial to impose a temporal constraint between when a region of space is seen, and when it is swept. Expressed temporally, V-Safety Constraint says "look before sweep". Some perception systems might take time to incorporate a measurement into their maps. We might also want to ensure a minimum time gap, to account for processing delays in a perception subsystem. We also might want to minimize the maximum time gap between when a region was seen and when it was swept. This is a heuristic to handle dynamic environments.

5 Data Structures for Efficient Planning

This chapter is joint work with Sathwik V. Karnik.

In this chapter, we explain how to use *persistent data structures* to accelerate planning problems. The key idea is to do structure sharing across multiple instances of hierarchical spatial data structures. We re-introduce Visibility-Aware Motion Planning.

5.1 Introduction

Motion planning is a crucial computation for many robotic systems that often requires significant resources. Spatial data structures provide asymptotic complexity benefits for many geometric problems, and used judiciously, they also produce practical benefits. In this chapter we develop a fully persistent spatial data structure (FPSDS) and explore its use in accelerating motion planning. Before introducing the FPSDS, we discuss two uses of *emphemeral* (not persistent) spatial data structures in robotics.

5.2 Traditional applications of spatial data structures

Rapidly-exploring Randomized Tree (RRT) planners typically use a nearest-neighbor (NN) data structure that supports dynamic insertions as planning progresses [80]. Note that the data structure corresponds to the "algorithmic" state of the RRT search procedure itself, not the state in the sense of the dynamics of the planning problem.

Physics simulations and ray tracing require collision checks between many entities. The entities are approximated with bounding volumes within a spatial data structure ("broad-phase collision checks") [81]. If the bounding volumes intersect, we say that the entities *interfere*. If two entities do not interfere, then they do not collide. As the simulation progresses, the bounding volumes are updated.

In both cases, the data structure is dynamically updated *destructively* (i.e. in place) since it is not necessary to access or modify previous versions of the data structure.

There are, however, many important formulations of motion planning problems that benefit from the use of spatial data structures that support access and modification of previous versions. Like in typical search, e.g. A* or RRT, this formulation has search nodes organized in a tree. Unlike other formulations, these search nodes are not "self-contained" in that they do not explicitly represent the full state. Computing successor search nodes and their priorities in a queue, in general, requires information not just from a given search node, but the entire path to the root node. We call this formulation *path-dependent*. A path-dependent formulation is beneficial in practice for problems where it is more efficient to represent the state implicitly as a sequence, e.g., of previously visited configurations, than it is to represent the state explicitly, e.g., with an occupancy grid or a probability distribution that aggregates information obtained along the path.

Path-dependent formulations lack the property of optimal substructure, without which an optimizing search must, in general, maintain multiple paths to each configuration. This situation also arises when the path cost is non-additive, and efficient solutions are possible in settings with an effective domination criterion [3].

A naïve approach to computing node successors in a path-dependent formulation will require time linear in the length of the path to the node, whereas in a typical search this operation takes (roughly) constant time. We can improve upon this by noting that, even though path-dependent formulations generally require information from the entire path, there are applications where only nodes that are local in the workspace are relevant. Conceptually, we require a spatial data structure at each search node*, though in order to benefit (asymptotically and in practice), we must leverage computation and data reuse. Our strategy is to use an FPSDS, which allows access and modification to any version of the spatial data structure [82]. For spatial data structures such as kd-trees [83], we may use the FPSDS to perform, for instance, range queries to a particular version of the spatial data structure to filter points within a bounding volume.

In this chapter, we describe how to apply this strategy to several problems and describe in detail our contribution of a specific type of FPSDS – a fully persistent nearest neighbor tree (FPNNT). We describe an application to Visibility-Aware Motion Planning (VAMP) [68], along with experimental results demonstrating the effectiveness of the FPSDS in illustrative domains. This application has some

* The spatial data structure corresponds to the problem state, not the algorithmic state as is the case in RRT

overhead, but we show a substantial performance benefit for large domains.

5.3 Motion Planning Applications

In this section, we outline three important classes of motion planning problems that benefit from path-dependent formulations.

Minimum Constraint Removal

In the Minimum Constraint Removal (MCR) [4] problem, the objective is to find a solution that may be infeasible, but that can be made feasible by the removal of a set of constraints. In general, one seeks a solution that requires removal of as few constraints as possible. Such problems require keeping track of the violations that have been accumulated along each path during the search.

Whereas in shortest-path motion planning, the state is simply the configuration, in MCR, the state of the search problem is the configuration and the set of constraint violations. So, although in shortest-path motion planning, it is sufficient to consider only a shortest path to a given configuration, MCR planning may need to consider multiple paths to a given configuration, each with a different violation set. Whereas the cost function (on configurations) in shortest-path motion planning is additive, in MCR, it is not, because we must not double-count constraint violations.

If there are many removable obstacles, then representing this set explicitly in the state can become prohibitive in memory and in time, since the violation set must be updated non-destructively, and therefore, it must be copied for the new descendants added to the node. For motivation and illustration, consider a variant of MCR, where the objective involves the path swept volume. Given a discretization of the workspace, "Minimum Swept Volume" planning is simply MCR, where each workspace cell (voxel) is a removable obstacle. In this setting, it is clear that explicitly representing an occupancy grid at each search node is prohibitive. MCR is amenable to the path-dependent formulation since a path of configurations induces a swept region.

Let n_i represent a search node in the search tree. Let $|S(\mathrm{n}_1, \ldots, \mathrm{n}_i)|$ denote the size of the swept region induced by the path of configurations represented in n_1 to n_i. We are interested in computing this

quantity and improving on the straightforward approach with time linear in the path length.

We provide a sketch of the improvement. For brevity, let $A_i = S(\mathrm{n}_1, \dots, \mathrm{n}_i)$ and $B_{i+1} = S(\mathrm{n}_i, \mathrm{n}_{i+1})$. To update the swept region from n_i to n_{i+1}, we can recursively compute $|A_{i+1}| = |A_i| + |B_{i+1} \setminus A_i|$. We must therefore compute the incremental swept volume B_{i+1}, but since we would like to avoid storing an explicit representation of A_i (we only need its size), we must visit each node and determine the incremental swept volume at that node, and subtract that set from B_{i+1}. This requires only temporary use of a workspace occupancy grid. However, this solution still takes time linear in the path length.

We improve upon this solution by using the FPSDS to compute fewer swept volumes along a path. In this case, we do not need to explicitly compute the entire swept volume A_i; rather, we only need to compute swept regions for transitions that interfere with B_{i+1}. These transitions can be determined with a range query, based on B_{i+1}, to the FPSDS that stores a workspace point corresponding to each transition along the path.

Belief-Space Planning

In the previous example, we relied on representing a swept volume explicitly as an occupancy grid or implicitly from a sequence of configurations. In a Partially-Observed Markov Decision Process (POMDP) [84], a belief state can be represented explicitly as a probability distribution or implicitly as a starting belief and history of actions and observations.

In general, only an approximation of an explicit representation of the belief state is possible. When the state includes the position of objects, and there are "complicated" constraints (e.g., objects are known to not penetrate), or "complicated" observations (e.g., some region of space is unoccupied), committing to an explicit posterior and using it in a recursive filtering strategy may "lock in" errors in the approximate representation [85]. The alternative is to store all observations, and perform inference on that data as needed.

Consider observations that indicate that the position of an object is uniformly distributed within a disk. The posterior can be computed by set intersection of the observation disks. Whereas the sequence of observations has a straightforward representation, an explicit exact representation of the intersection region does not. However, given

the observations, we may produce samples of the exact "posterior" by e.g. rejection sampling.

Observations are generated by the environment, but during belief-space planning we might approximate by choosing the maximum-likelihood observations [86]. The belief over object position could then be used to determine the collision-free probability of an incremental motion. However, only a subset of observations is likely to be relevant to determining such a quantity. Under certain realistic assumptions, a range query to the FPSDS returns the relevant observations.

Visibility-Aware Motion Planning

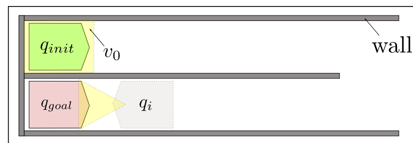


Figure 5.1: Example VAMP problem instance. In the domains we discuss, the viewcone (in yellow) is fixed relative to the robot.

In the most classic formulation of robot motion planning, the geometry of the robot and environment is fully determined, and there is perfect actuation, so the problem of finding a path in configuration space that avoids obstacles can be solved "open-loop." We have previously introduced *visibility-aware motion planning*, or VAMP [68], where there is uncertainty about the environment in that there may be obstacles not represented in the map given to the motion planner. The motion planner must produce an open-loop path that avoids all obstacles represented in the map *and* is safe with respect to "unknown" obstacles. At each moment, the robot may move into a region of space only if there is no obstacle in that region *and* that region was observed (by an on-board sensor) earlier in the path. The first condition can be determined with collision checking of an articulated body against a static obstacle map. The second condition – called the *visibility constraint* – requires a different computation and contributes substantially to the planning time. Figure 5.1 shows an example of a VAMP problem instance in which a violation-free path exists and requires the robot to view the lower hallway before moving backwards into the goal. In the remainder, we address algorithmic optimizations to this computation using a FPSDS.

5.4 Fully Persistent Nearest-Neighbor Tree

In this section, we discuss our contribution of the fully persistent nearest-neighbor tree (FPNNT), which stores its points efficiently

with a tree. The approach is inspired by the static-to-dynamic logarithmic method [87], which organizes a collection of static trees. For n points, there is a static tree that contains $2^i M$ points iff the ith bit of the binary representation of $\lfloor (n-1)/M \rfloor$ is "1," and $(n-1) \pmod{M} + 1$ points in the "remainder," which has maximum size equal to a fixed parameter M. In the FPNNT, the static trees are nearest-neighbors (NN) [88] trees.

Each node in the FPNNT stores (1) a new point and label in the remainder, (2) a pointer to the parent node, (3) the number of most recent predecessors (corresponding to the size of the remainder), and (4) a dynamically-sized array of pointers (the *history*) to NN trees. An example excerpted FPNNT is shown in Figure 5.2.

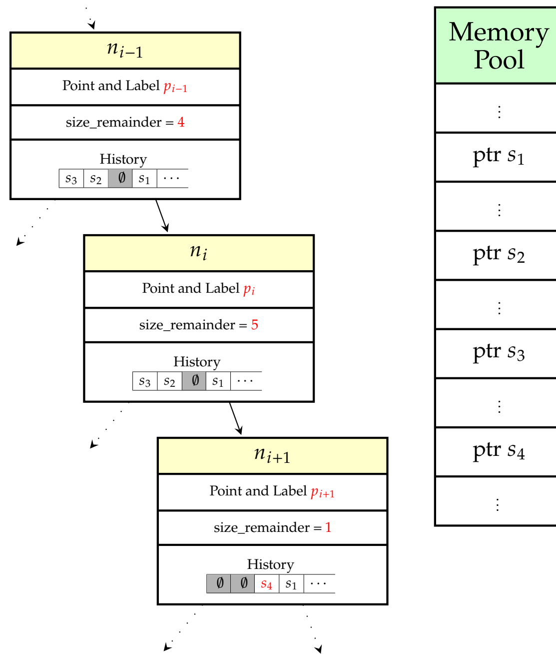


Figure 5.2: Example of part of a fully persistent tree with information stored at each node. The red text at each node denotes the changes from its parent node. The subsequence of pointers – s_1, s_2, s_3, and s_4 – in the persistent tree nodes is shown in the order of allocations in the memory pool. Here, $M = 5$.

Insertion

Algorithm 8 shows the pseudocode for the function Insert_Node, which creates a new node of the FPNNT that corresponds with a version of the data structure containing the new point-label pair p_{new} and all the points in n_{par}. In the insertion, there are two cases: $rem_{par} < M$ and $rem_{par} = M$. In the first case, the node can simply be inserted. In the second case, we first call the Remainder function, which collects the point-label pairs from the rem_{par} most recent predecessors. In addition, we iterate through the pointers in the node history h_{new}. In particular, we collect the $2^i M$ points stored in each of the trees T_i in the history for $0 \leq i \leq k-1$, where T_k

is the first tree that is empty or null. Altogether these $2^k M$ points are denoted as pt_labels. An NN tree is batch-constructed from these points. Finally, we update h_{new} so that the first k pointers are null pointers and the $(k+1)^{th}$ element is the pointer to the newly constructed NN tree. The newly computed rem_{new} and h_{new} are then used to construct the new node, which is then inserted into the FPNNT with parent n_{par}.

Algorithm 8 INSERT_NODE(n_{par}, p_{new})

```
 1: h_new ← n_par.history
 2: rem_par ← n_par.size_remainder
 3: t_new ← ∅
 4: if rem_par ≥ M then
 5:     pt_labels ← REMAINDER(n_par)
 6:     i ← 0                                          ▹ Index into history
 7:     while h_new[i] ≠ ∅ do
 8:         pt_labels ← APPEND(pt_labels, h_new[i].pt_labels)
 9:         h_new[i] = ∅
10:         i ← i + 1
11:     end while
12:     t_new ← BUILD_NN_TREE(pt_labels)
13:     h_new[i] ← t_new
14: end if
15: rem_new ← rem_par + 1 (mod M)
16: return CONSTRUCT_NODE(rem_new, h_new, p_new)
```

Figure 5.2 illustrates both cases of the insertion at a given node. Specifically, the changes in the information stored from node to child node are denoted in red. The first case is illustrated in the insertion of p_i at node n_i with parent n_{i-1}. The only changes from n_{i-1} to n_i include the new point and label and the size of the remainder. In the second case of insertion, as seen in the insertion of p_{i+1} at node n_{i+1} with parent n_i, we observe the same changes as in the first case, as well as the new NN-tree pointer s_4 in the history.

Range Query

In the RANGE_QUERY function, we search for the relevant point-label pairs such that the points are within a bounding ball with radius r_{query} of and center w. To do so, we iterate through h_{cur} and perform the standard range query in each NN tree, and we iterate through the remainder rem_{cur} of n_{cur} and perform a brute-force range query.

Complexity Analysis

In this section, we evaluate the time complexities of the INSERT_NODE and RANGE_QUERY functions. Assume that the NN trees used in the FPNNT are kd-trees. Let N be the total number of nodes in the persistent tree, and let L be the length of the longest path in the tree from the root. Let D be the dimension of the points (in our case, the dimension of the workspace).

With the motion planning application solutions we seek to optimize, we keep a bounded number of paths to a given configuration. Thus, we can assume that the number of nodes on a given depth in the tree grows polynomially, not exponentially.

Insertion Time

For search trees where the number of nodes per level grows polynomially, the time complexity of each insertion is amortized $O((\log L)^2)$.

Range Query Time

Assuming that we query in the longest path of length L, we must perform the range query in the remainder and for each of the kd-trees stored in the history of the leaf node. For the remainder, the time complexity is constant, as M is a constant. For the largest of the kd-trees, the time complexity is worst-case $O(L^{1-1/D})$, from the orthogonal range query time complexity [83]. Let K be the total number of points within the radius of the query point. In total, we have a time complexity of $O(L^{1-1/D} \log L + K)$, where we use the fact that there are $O(\log L)$ trees.

Comparison with Baseline

In the baseline method, at each node, we only store a point-label pair and perform brute-force range queries along the path from a node to the root.

Operation	Baseline	FPNNT
INSERT_NODE	$O(1)$	$O((\log L)^2)$
RANGE_QUERY	$O(L)$	$O(L^{1-1/D} + K)$

Table 5.1: Comparison of amortized time complexities.

We can see from Table 5.1 that the FPNNT provides a much faster amortized asymptotic runtime in the FPNNT RANGE_QUERY than the baseline RANGE_QUERY. Although this optimization does result in a slower amortized insertion time, in our experiments, we show that this can be a favorable trade off. Note that in our planning applications, the number of insertions and queries are roughly equal.

5.5 VAMP Problem Formulation

To observe the benefits of the FPSDS, we focus on the application to VAMP. We now provide a formulation of VAMP similar to the one provided in [68].

Let W be the workspace ($\mathbb{R}^2$ or $\mathbb{R}^3$), and let C be the configuration space of the robot. Furthermore, let $W_{obs} \subseteq W$ be the region of space known to contain obstacles. Let q_0 be the initial configuration and let $v_0 \subseteq W$ be the initial visible region. In this problem, we assume that the entire space swept from the motion of the robot during its path must be previously viewed but the new visible regions are gained only at the end of each primitive motion.

We now define the following functions characterizing visible regions and swept volumes. Let $\mathbb{P}(X)$ denote the power set of the set X. We define $V : C \to \mathbb{P}(W)$ to be the visibility function – that is, the function computes the subset of W that is visible from a configuration. We overload the notation and define $V([q_1, \ldots, q_n]) = \bigcup_{i=1}^{n} V(q_i)$. Let $S : C \to \mathbb{P}(W)$ denote the swept volume function. As before, we extend this definition so that $S(q_i, q_j) \subseteq W$ represents the space the robot sweeps when moving from q_i to q_j. More generally, we define $S([q_1, \ldots, q_n]) = \bigcup_{i=1}^{n-1} S(q_i, q_{i+1})$. Finally, let $Q_{goal} \subseteq C$ be a set of goal configurations. A VAMP problem instance is represented by the tuple $(W, C, V, S, W_{obs}, q_0, Q_{goal}, v_0)$.

We assume a graph of primitive motions, with vertices embedded in C. If there is an edge between q_i and q_j, and $S(q_i, q_j) \cap W_{obs} = \emptyset$, then it is collision-free. A path $[q_1, \ldots, q_n]$ is said to be *feasible* if and only if: (1) each edge from q_i to q_{i+1} in the path is collision-free and (2) the path satisfies the *visibility constraint* – that is, $S(q_i, q_{i+1}) \subseteq v_0 \cup V([q_1, \ldots, q_i])$ for all $i \in \{1, \ldots, n-1\}$. The state space of this problem is naturally $(q, v) \in C \times \mathbb{P}(W)$, representing the configuration and the visible region attained along the path to the configuration. As in Section 5.3, the visible region v_i can be represented implicitly by an initial region v_0 and path $[q_1, \ldots, q_i]$. Note that, due to path dependence, the visibility constraint cannot

be applied "pointwise" in the same way that the collision constraint can.

5.6 Relaxed VAMP Solution

A feasible solution to VAMP can be found in time polynomial in the size of the problem description. We conjecture that the optimal solution is hard, and seek approximate solutions. The strategy is to relax the visibility constraint and keep track of the *unseen swept region* $\bigcup_{i=1}^{n} S(q_i, q_{i+1}) \setminus (v_0 \cup V([q_1, \ldots, q_i]))$, also referred to as the (visibility) violation region. We refer to the computation of the unseen swept region as a *visibility query*.

This relaxed VAMP search produces a collision-free path that penalizes, but ultimately allows, visibility violations. As such, it is not responsible for producing a feasible solution on its own, but the violation region is used to inform another search procedure described in earlier work [68], which uses the region as a visibility subgoal. Note that minimizing the size of the violation region does not guarantee that the final path is optimal in length, but it is a useful heuristic for generating effective subgoals. We will ultimately minimize an over-approximation to the size of the violation region.

We avoid storing a representation of the visible region and violation region at each search node, and determine the incremental violation region, and accumulate its size. Determining the incremental violation region can be done inefficiently by iterating towards the root node of the search tree until the incremental unseen region is empty, or until the root node is reached. Due to the relaxation of the visibility constraint, there are some domains where traversing to the root occurs often (e.g. Figure 5.3).

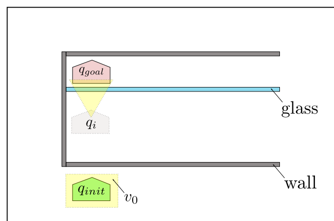


Figure 5.3: Motivating example of a VAMP problem instance in which computing the unseen swept region may be expensive. The violation-free path requires the robot to look through the glass wall into the hallway containing q_{goal}. The path found from the relaxed VAMP problem results in unseen swept regions.

The size of the violation region of an incremental motion – later described in Algorithm 9 – is used to determine the transition cost in the relaxed VAMP search (corresponding to the VAMP_PATH_VIS algorithm with relaxed = true in earlier work [68])). This results in an over-approximation of the violation region size, but means that

only a subsequence of the search nodes along the path to the root are relevant for determining the new cost, which enables the key optimization of this chapter.

5.7 Efficient Visibility Queries

The visibility queries can be made more efficient by filtering out configurations outside a bounding volume. We formalize the use of bounding volumes in the next section.

Bounding Volumes

Let $\mathbf{dist}(p_1, p_2)$ denote the Euclidean distance between two points p_1 and p_2. We define a ball $\mathcal{B}(c, r)$ to be the set of points p such that $\mathbf{dist}(p, c) \leq r$. Let r_{vis} be the radius of the smallest ball containing any visible region. We define $\varphi : C \to W$ such that $\varphi(q) = w_v$, where $V(q) \subseteq \mathcal{B}(w_v, r_{vis})$. Furthermore, define $\psi : C \times C \to W \times \mathbb{R}_{\geq 0}$ such that $\psi(q_a, q_b) = (w_s, r_s)$, where r_s is the smallest radius such that $S(q_a, q_b) \subseteq \mathcal{B}(w_s, r_s)$. Denote $r_{query} := r_{vis} + r_s$.

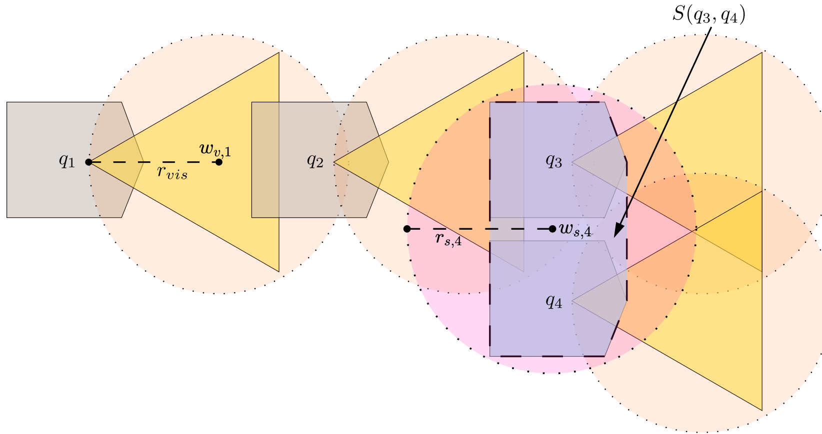


Figure 5.4: Trajectory $[q_1, q_2, q_3, q_4]$ with swept region $S(q_3, q_4)$ from q_3 to q_4. The viewcones $V(q)$ are shown with bounding radius r_{vis}. For $V(q_1)$, we show that the bounding ball has center $w_{v,1}$. The swept region $S(q_3, q_4)$ is shown with a bounding ball $\mathcal{B}(w_{s,4}, r_{s,4})$.

Consider the path $[q_1, \ldots, q_i]$ in our search tree, and suppose we are interested in calculating the unseen swept volume $S(q_i, q_{i+1}) \setminus V([q_1, \ldots, q_i])$ for some configuration q_{i+1}. Let $\psi(q_i, q_{i+1}) = (w_{s,i+1}, r_{s,i+1})$. For $r_{query} = r_{vis} + r_{s,i+1}$, we can guarantee that all configurations q for which $\mathbf{dist}(\varphi(q), w_{s,i+1}) > r_{query}$ have the property that $S(q_i, q_{i+1}) \cap V(q) = \emptyset$. Figure 5.4 illustrates a trajectory with swept region $S(q_3, q_4)$ from q_3 to q_4. It can be seen that if the distance between the centers of the viewcone bounding balls and $\mathcal{B}(w_{s,4}, r_{s,4})$ exceeds $r_{vis} + r_{s,4}$, the balls do not intersect and, thus, $S(q_3, q_4) \cap V(q) = \emptyset$.

Since computing the incremental updates to the unseen swept region (see Algorithm 9 Line 6) can be expensive, our approach considers only the configurations in the path that do not interfere with $\mathcal{B}(w_{s,i+1}, r_{s,i+1})$. Algorithm 9 illustrates this point through the call to RANGE_QUERY in the FIND_VIS_VIOL function, which returns $S(q_i, q_{i+1}) \setminus V([q_1, \dots, q_i])$.

Algorithm 9 FIND_VIS_VIOL(path, q_{i+1}, r_{vis})

```
1:  q_i ← path[end]
2:  (w_{s,i+1}, r_{s,i+1}) ← ψ(q_i, q_{i+1})
3:  r_query ← r_vis + r_{s,i+1}
4:  unseen_swept ← S(q_i, q_{i+1})
5:  for q_j ∈ RANGE_QUERY(path, w_{s,i+1}, r_query) do
6:      unseen_swept ← unseen_swept \ V(q_j)
7:      if unseen_swept = ∅ then
8:          break
9:      end if
10: end for
11: return unseen_swept
```

RANGE_QUERY can be determined by brute force, requiring time linear in the path length. We can improve on this, at the cost of storage, by performing range queries using the FPNNT in the VAMP application. Specifically, we use kd-trees in the BUILD_NN_TREE function of Algorithm 8. Using the notation of Section 5.4, we can set the "points" as viewcone bounding ball centers $w_{v,i}$ and "labels" as configurations q_i. Thus, RANGE_QUERY returns the configurations with viewcone bounding balls interfering with the swept region of interest.

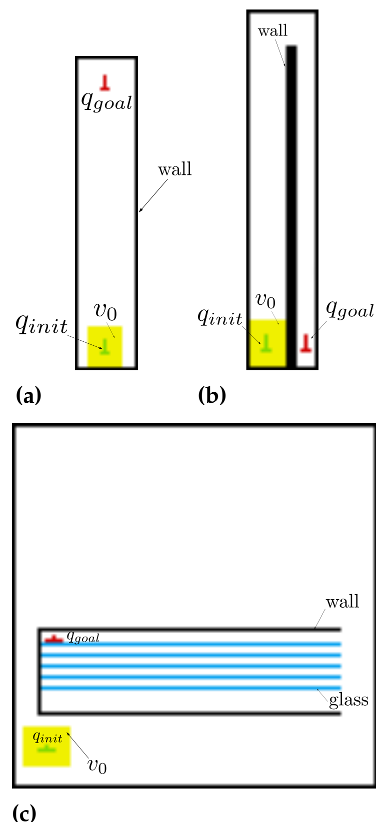


Figure 5.5: Experiment domains: (a) ONEHALLWAY, (b) HORSESHOEHALLWAY, and (c) GLASSHALLWAY.

5.8 Experiments

We now discuss the experiments that demonstrate that the FPNNT, using kd-trees, provides an efficient way to perform the RANGE_-QUERY in Algorithm 9. We measure the performance of solving relaxed VAMP problems and compare the FPNNT with the baseline of performing brute-force range queries. The experiments are implemented in the Julia language [89]. Here, we omit the compilation times.

Throughout our experiments, we focus on domains with a discretized workspace and configuration space. The workspace is planar and the configuration space represents the position of a reference point of the robot, along with the robot's orientation. There

are 6 possible actions, one in the positive and negative direction for each dimension of the configuration space, forming a 6-connected lattice in the configuration space. The viewcone is fixed and is about 1.5 times the length of the robot.

Experimental Results

For each of the domains shown in Figure 5.5, we parameterize the length of the domain for demonstrating asymptotic performance of the use of different visibility query optimization methods. For the OneHallway and HorseshoeHallway domains, we vary the vertical lengths from 1000 to 15000 cells, incremented by 1000 cells, while maintaining the widths of the domains. For the GlassHallway domain, we vary the sizes from 100×100 to 1500×1500 cells, incremented by 100 cells. Note that as we increase the size of the GlassHallway, the number of glass hallways linearly increases as a function of the length. In all of these domains, the hallway widths remain constant.

Figure 5.6 and Figure 5.7 show the comparisons of the use of the FPNNT with the baseline in terms of runtimes and search tree storage. We separate the plots for GlassHallway from those corresponding to the other domains to emphasize the difference in scaling domain lengths.

The left column of plots shows the comparisons of runtimes and the right column of plots shows the comparisons of search tree storage. For each domain length, we ran 10 iterations of the Vamp_Path_Vis algorithm for the baseline and the use of the FPNNT. We collected and reported the average total runtime and the average runtime spent in the Find_Vis_Viol function. Additionally, for each vamp problem instance and range query method, we reported the overall memory consumed by the search.

In Figure 5.6, we can see that in the simple vamp problem instance of the OneHallway domain, the total runtime of the search is approximately 1 second for even the highest domain sizes we tested with. We can see that the baseline method outperforms our method in this simple case. It is also apparent from the empirical data that Find_Vis_Viol takes the majority of the runtime, thus illustrating the importance of these performance improvements.

As the unseen swept region computation requires more visibility calculations, the domains – HorseshoeHallway and GlassHallway – illustrate a significant performance improvement. In the GlassHallway domain, we observe the performance improvement at even

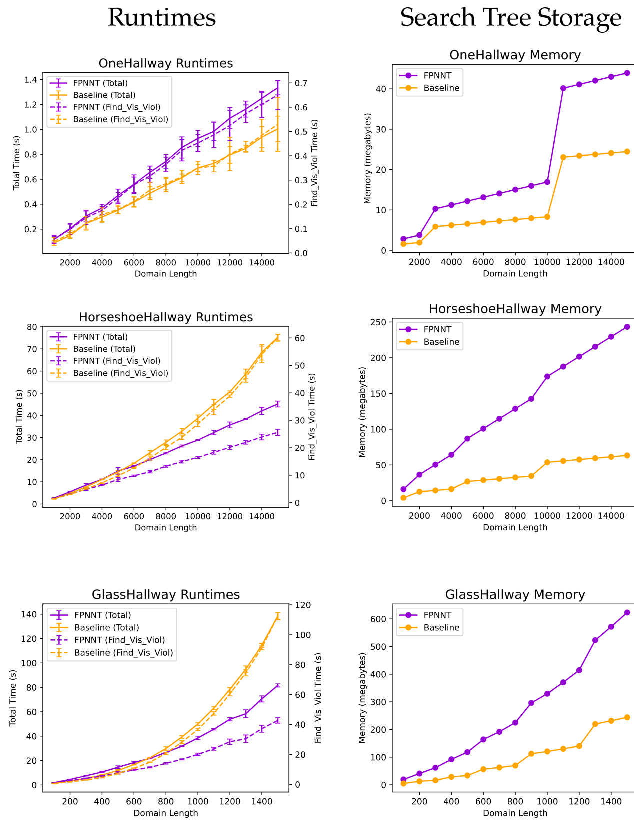


Figure 5.6: These plots show the results for the OneHallway and HorseshoeHallway domains. The left column shows the overall runtimes and total times spent in Find_Vis_Viol. The right column shows the memory storage (via `Base.summarysize`) of the search trees. Discontinuities in memory use as the length grows are attributed to Julia data structure implementation details [90].

Figure 5.7: These plots show similar statistics as shown in Figure 5.6 but for the GlassHallway domain.

smaller sizes. The overall runtime improvement does not exactly match the reduction in time spent in Find_Vis_Viol due to the insertion time into the FPNNT. The insertion times are already accounted for in the total runtimes.

The second column of plots in Figure 5.6 and Figure 5.7 shows the total memory of the search trees. Since each node of the FPNNT is stored at each node of the search tree, our method is expected to have a search tree that costs more memory than the baseline. Across all of the domains, the search tree storage plots reflect this expectation. However, for the domains tested, the ratio of memory stored in the search tree between our method and the baseline method remains under 4.2 even as the domain size grows.

5.9 Discussion

Fully persistent data structures are key to some algorithms [91], and find applications in strictly functional programming, where mutation is prohibited [92]. To our knowledge, this is the first use of a fully persistent data structure to accelerate planning. We showcase

the approach in discrete search on a lattice, but the idea extends to any tree-based approach, such as RRT or searching for paths within a Probabilistic Road Map (PRM). In the applications we discussed, the FPSDS accelerates what is otherwise a linear-time search to determine points in a range, where these points represent the centers of bounding balls of predetermined radius. For large instances of a recent formulation called Visibility-Aware Motion Planning, we find that the FPNNT improves the planning runtimes by a factor of 2, at the expense of memory use. Other choices of FPSDS, such as using an R-Tree [93] to represent the extent of bounding volumes, may further improve the performance in various settings.

6 Discussion

6.1 Execution

In the previous chapters, we provide results on planning on a given environment. The reason to plan, however, is to execute plans. And the reason for VAMP is to ensure that critical regions are observed, in case there are unforeseen obstacles, so that an execution monitor can replan as necessary. In this chapter, we demonstrate replanning in an environment that is observed incrementally.

6.2 Issues arising in replanning

Commitment

Take the situation depicted in Figure 6.1. Assume that the robot can only detect if there is additional wall by being at the known end of the wall. There are three critical configurations, q_0 (depicted with label init), q_a (the robot is at the known end of the horizontal wall), and q_b (the robot is at the known end of the vertical wall). Let (a, b) describe the known model of the environment. There are always two conceivable paths, one around the horizontal wall, and the other around the vertical wall.

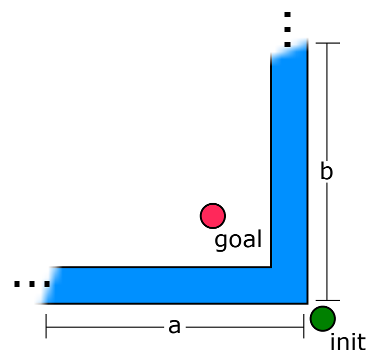


Figure 6.1: In this environment, there are two walls that are detected to have length a and b. The walls continue beyond the figure, possibly indefinitely. If both walls are infinitely long, then there is no solution path from the initial to goal configuration.

- From q_0, the paths have lengths $2a$ and $2b$ respectively.
- From q_a, the paths have lengths a and $a + 2b$ respectively.
- From q_b, the paths have lengths $b + 2a$ and b respectively.

If initially $(a, b) = (1, 1)$, then the path lengths are 2 and 2. If the planner is optimal, it may choose either path. For sake of illustration, suppose it chooses the first path, and once it gets to q_a, it discovers that the wall longer. The environment is now known to be $(2, 1)$, and the lengths of the paths from q_a are 2 and 4. Replanning from this configuration will generate a plan that continues along a. Now, if the horizontal wall is an infinite obstacle, but the vertical wall is finite, then this situation will continue indefinitely, and the robot will never achieve the goal despite it being reachable. This problem does not arise under the assumption that the environment is bounded in size. We think of this as a strategy with very high commitment.

Suppose both walls are indeed finite and that the robot switches paths as soon as it detects that the current wall is longer than was

modeled. Let a_i and b_i denote the known length of the wall on the ith visit. For example if the robot travels down the a wall first, and it's the shorter wall with length n, then the accumulated path length is $2a_1 + 2b_1 + 2a_2 + 2b_2 + \cdots + 2a_k$. If the other wall is shorter, then there is an extra $2b_k$ term. k is the number of times the robot goes down the first wall. We will consider symmetric strategies ($a_i = b_i$), so on average the accumulated path length is $4(a_1 + a_2 + \cdots + a_{k-1}) + 3a_k$.

Consider a strategy with very low commitment. As soon as a wall is detected to be longer than anticipated, the robot switches the path. In our notation, $a_i = b_i = i$ and $k = n$. The expected total path length is $2n^2 + n$ where n is the length of the shortest wall. The shortest path given complete information is $2n$, so this strategy has an expected decision-theoretic regret of $2n^2 - n$.

$1 + 2 + 3 + \cdots + n = n(n+1)/2$

If the length of the longer wall is m, and you commit to going down that wall, then the total path is $2m$. This strategy in this situation has a regret of $2(m - n)$. The expected regret is $m - n$.

What about a multiplicative strategy? If a wall is longer than expected, the robot does not immediately switch to the other wall, but instead continues to a depth twice as much as it tried last time. In our notation, $a_i = b_i = 2^i$ and $k = \log_2 n$ For the sake of simplicity assume the true wall lengths make k an integer. On average, the accumulated path length is $4(2^1 + 2^2 + \cdots + 2^{k-1}) + 3(2^k)$ Or, $4(2^k) + 3(2^k)$, which is $7n$. The expected regret is therefore $5n$.

$1 + 2 + 4 + \cdots + 2^i = 2^{i+1} - 1$

This problem of *thrashing* is seemingly inescapable when performing goal-seeking behavior in an unknown environment. Intuitively, if the robot gives up too early, it will thrash too much. Unfortunately, this can also arise when using a suboptimal planner, as is the case for our practical algorithms for VAMP.

6.3 Replanning

When an observation is made that invalidates the current plan, we can choose to immediately replan, or we could execute the plan up until the point it is no longer valid. In the following experiments, we immediately replan, but because the planner is suboptimal, it thrashes. As demonstrated above, some intermediate amount of commitment to the invalidated plan can reduce regret.

We first demonstrate replanning when the robot has very little knowledge about the environment in Figure 6.2. In such a situation, it is expected that a more efficient strategy is to make a map using methods discussed in Section 2.2. In the domain illustrated in

Figure 6.2, the planner is careful to check the hallway before going backwards into it. However, if there truly is something in the way, there is no other path to the goal. In a domain where there are multiple paths to the goal, the replanning could succeed. It is better for planning to fail (the robot can ask for help, or perhaps move the obstacle) than to cause an unexpected collision.

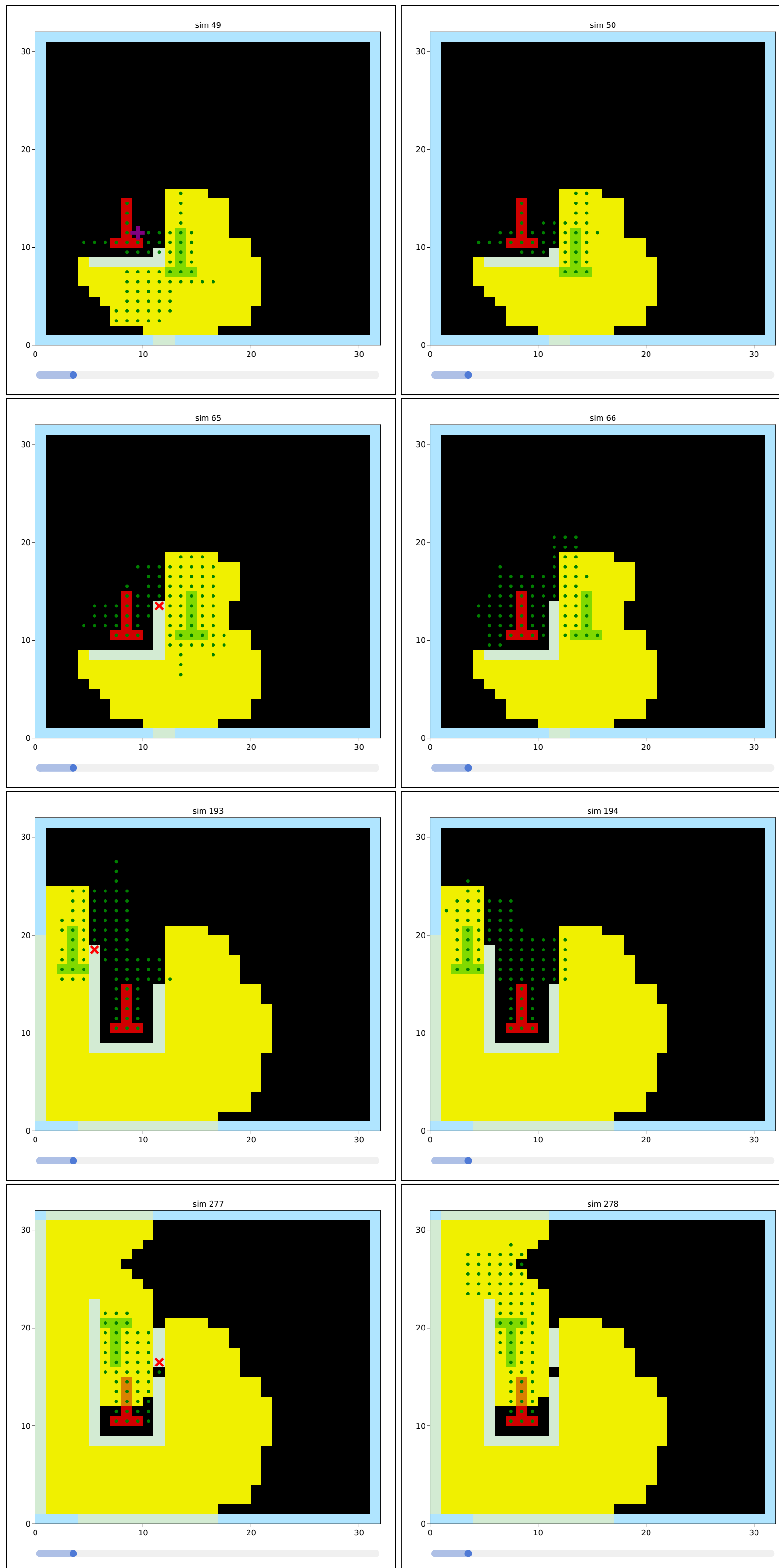


Figure 6.2: Illustration of replanning starting from no knowledge of the environment except for its extent (the border) in the discrete HallwayHard domain. The red robot depicts the configuration goal. The green dots represent the swept volume of the current plan. The plan is executed until an obstacle is in the way of the robot (a red cross), or an occluder blocks a necessary view (a purple plus), at which point a new plan is made.

Bibliography

[1] Steven M. LaValle. *Planning Algorithms*. Cambridge: Cambridge University Press, 2006. (Visited on 11/28/2021) (cited on pages 12, 22, 52).

[2] Dave Ferguson, Maxim Likhachev, and Anthony Stentz. 'A Guide to Heuristic-based Path Planning'. In: *Proceedings of ICAPS '05 Workshop on Planning under Uncertainty for Autonomous Systems*. June 2005, p. 10 (cited on page 15).

[3] Oren Salzman, Brian Hou, and Siddhartha Srinivasa. 'Efficient Motion Planning for Problems Lacking Optimal Substructure'. In: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017.* Ed. by Laura Barbulescu et al. AAAI Press, 2017, pp. 531–539 (cited on pages 16, 40, 77).

[4] Kris Hauser. 'The Minimum Constraint Removal Problem with Three Robotics Applications'. In: *The International Journal of Robotics Research* 33.1 (Jan. 1, 2014), pp. 5–17. DOI: 10.1177/0278364913507795. (Visited on 09/03/2021) (cited on pages 16, 40, 78).

[5] Brian Axelrod, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. 'Provably Safe Robot Navigation with Obstacle Uncertainty'. In: *The International Journal of Robotics Research* (June 7, 2018). DOI: 10.1177/0278364918778338. (Visited on 03/12/2021) (cited on pages 16, 28).

[6] Nicholas M Stiffler and Jason M O'Kane. 'Complete and Optimal Visibility-Based Pursuit-Evasion'. In: *The International Journal of Robotics Research* 36.8 (July 1, 2017), pp. 923–946. DOI: 10.1177/0278364917711535. (Visited on 03/12/2021) (cited on pages 16, 22, 40, 49, 64).

[7] A. Bry and N. Roy. 'Rapidly-Exploring Random Belief Trees for Motion Planning under Uncertainty'. In: *2011 IEEE International Conference on Robotics and Automation*. 2011 IEEE International Conference on Robotics and Automation. May 2011, pp. 723–730. DOI: 10.1109/ICRA.2011.5980508 (cited on pages 16, 28, 40, 47, 49).

[8] Brendan Englot and Franz S. Hover. 'Sampling-Based Coverage Path Planning for Inspection of Complex Structures'. In: *Proceedings of the Twenty-Second International Conference on International Conference on Automated Planning and Scheduling*. ICAPS'12. Atibaia, São Paulo, Brazil: AAAI Press, June 25, 2012, pp. 29–37 (cited on pages 20, 22, 28).

[9] Vladimir J. Lumelsky and Alexander A. Stepanov. 'Path-Planning Strategies for a Point Mobile Automaton Moving amidst Unknown Obstacles of Arbitrary Shape'. In: *Algorithmica* 2.1 (Nov. 1, 1987), pp. 403–430. DOI: 10.1007/BF01840369. (Visited on 03/12/2021) (cited on pages 20, 22).

[10] A. Bircher et al. 'Receding Horizon "Next-Best-View" Planner for 3D Exploration'. In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016 IEEE International Conference on Robotics and Automation (ICRA). May 2016, pp. 1462–1468. DOI: 10.1109/ICRA.2016.7487281 (cited on pages 21, 24, 28, 29).

[11] 'Sensor Based Motion Planning: The Hierarchical Generalized Voronoi Graph'. In: *Algorithms for Robotic Motion and Manipulation*. Ed. by Jean-Paul Laumond and Mark Overmars. 0th ed. A K Peters/CRC Press, Feb. 11, 1997, pp. 59–74. DOI: 10.1201/9781439864524-10. (Visited on 03/26/2021) (cited on page 22).

[12] Enric Galceran and Marc Carreras. 'A Survey on Coverage Path Planning for Robotics'. In: *Robotics and Autonomous Systems* 61.12 (Dec. 1, 2013), pp. 1258–1276. DOI: 10.1016/j.robot.2013.09.004. (Visited on 03/12/2021) (cited on page 22).

[13] B. Davis, I. Karamouzas, and S. J. Guy. 'C-OPT: Coverage-Aware Trajectory Optimization Under Uncertainty'. In: *IEEE Robotics and Automation Letters* 1.2 (July 2016), pp. 1020–1027. DOI: 10.1109/LRA.2016.2530302 (cited on page 22).

[14] Sándor P. Fekete, Joseph S. B. Mitchell, and Christiane Schmidt. 'Minimum Covering with Travel Cost'. In: *Journal of Combinatorial Optimization* 24.1 (July 1, 2012), pp. 32–51. DOI: 10.1007/s10878-010-9303-0. (Visited on 03/15/2021) (cited on page 22).

[15] Esther M. Arkin, Sándor P. Fekete, and Joseph S. B. Mitchell. 'Approximation Algorithms for Lawn Mowing and Milling'. In: *Computational Geometry* 17.1 (Oct. 1, 2000), pp. 25–50. DOI: 10.1016/S0925-7721(00)00015-8. (Visited on 03/15/2021) (cited on page 22).

[16] Wei-pang Chin and Simeon Ntafos. 'Optimum Watchman Routes'. In: *Information Processing Letters* 28.1 (May 13, 1988), pp. 39–44. DOI: 10.1016/0020-0190(88)90141-X. (Visited on 03/12/2021) (cited on page 22).

[17] Brian P. Gerkey, Sebastian Thrun, and Geoff Gordon. 'Visibility-Based Pursuit-evasion with Limited Field of View'. In: *The International Journal of Robotics Research* 25.4 (Apr. 1, 2006), pp. 299–315. DOI: 10.1177/0278364906065023. (Visited on 03/12/2021) (cited on page 22).

[18] Zhan Wei Lim, David Hsu, and Wee Sun Lee. 'Adaptive Informative Path Planning in Metric Spaces'. In: *The International Journal of Robotics Research* 35.5 (Apr. 1, 2016), pp. 585–598. DOI: 10.1177/0278364915596378. (Visited on 03/15/2021) (cited on page 22).

[19] Steven M. LaValle. 'Sensing and Filtering: A Fresh Perspective Based on Preimages and Information Spaces'. In: *Foundations and Trends® in Robotics* 1.4 (Feb. 21, 2012), pp. 253–372. DOI: 10.1561/2300000004. (Visited on 12/05/2021) (cited on page 22).

[20] Thomas H Cormen et al. *Introduction to Algorithms*. MIT press, 2009 (cited on page 23).

[21] Nabil H. Mustafa, Rajiv Raman, and Saurabh Ray. 'Settling the APX-Hardness Status for Geometric Set Cover'. In: *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*. 2014 IEEE 55th Annual Symposium on Foundations of Computer Science. Oct. 2014, pp. 541–550. DOI: 10.1109/FOCS.2014.64 (cited on page 23).

[22] J. O'Rourke. *Art Gallery Theorems and Algorithms*. Vol. 57. Oxford, 1987 (cited on page 23).

[23] B. Yamauchi. 'A Frontier-Based Approach for Autonomous Exploration'. In: *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97. 'Towards New Computational Principles for Robotics and Automation'*. Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97. 'Towards New Computational Principles for Robotics and Automation'. July 1997, pp. 146–151. DOI: 10.1109/CIRA.1997.613851 (cited on pages 24, 27, 29).

[24] L. Heng et al. 'Efficient Visual Exploration and Coverage with a Micro Aerial Vehicle in Unknown Environments'. In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015 IEEE International Conference on Robotics and Automation (ICRA). May 2015, pp. 1071–1078. DOI: 10.1109/ICRA.2015.7139309 (cited on pages 24, 27).

[25] G. Oriolo et al. 'The SRT Method: Randomized Strategies for Exploration'. In: *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*. IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004. Vol. 5. Apr. 2004, 4688–4694 Vol.5. DOI: 10.1109/ROBOT.2004.1302457 (cited on page 24).

[26] Christian Dornhege and Alexander Kleiner. 'A Frontier-Void-Based Approach for Autonomous Exploration in 3D'. In: *Advanced Robotics* 27.6 (Apr. 1, 2013), pp. 459–468. DOI: 10.1080/01691864.2013.763720. (Visited on 03/12/2021) (cited on pages 24, 28, 29).

[27] K. E. Bekris and L. E. Kavraki. 'Greedy but Safe Replanning under Kinodynamic Constraints'. In: *Proceedings 2007 IEEE International Conference on Robotics and Automation*. Proceedings 2007 IEEE International Conference on Robotics and Automation. Apr. 2007, pp. 704–710. DOI: 10.1109/ROBOT.2007.363069 (cited on pages 24, 28, 29).

[28] Mikko Lauri and Risto Ritala. 'Planning for Robotic Exploration Based on Forward Simulation'. In: *Robotics and Autonomous Systems* 83 (Sept. 1, 2016), pp. 15–31. DOI: 10.1016/j.robot.2016.06.008. (Visited on 03/12/2021) (cited on pages 24, 28).

[29] Frank Hoffmann et al. 'The Polygon Exploration Problem'. In: *SIAM Journal on Computing* 31.2 (Jan. 1, 2001), pp. 577–600. DOI: 10.1137/S0097539799348670. (Visited on 03/12/2021) (cited on page 24).

[30] Sándor P. Fekete and Christiane Schmidt. 'Polygon Exploration with Time-Discrete Vision'. In: *Computational Geometry* 43.2 (2010), pp. 148–168. DOI: 10.1016/j.comgeo.2009.06.003 (cited on page 24).

[31] C. Cadena et al. 'Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-Perception Age'. In: *IEEE Transactions on Robotics* 32.6 (Dec. 2016), pp. 1309–1332. DOI: 10.1109/TRO.2016.2624754 (cited on page 24).

[32] Lucas Janson, Tommy Hu, and Marco Pavone. 'Safe Motion Planning in Unknown Environments: Optimality Benchmarks and Tractable Policies'. In: Robotics: Science and Systems XIV. Vol. 14. June 26, 2018. (Visited on 03/12/2021) (cited on pages 24, 28).

[33] Ioannis Arvanitakis, Anthony Tzes, and Konstantinos Giannousakis. 'Synergistic Exploration and Navigation of Mobile Robots under Pose Uncertainty in Unknown Environments'. In: *International Journal of Advanced Robotic Systems* 15.1 (Jan. 1, 2018), p. 1729881417750785. DOI: 10.1177/1729881417750785. (Visited on 03/12/2021) (cited on pages 24, 28).

[34] Charles Richter, William Vega-Brown, and N. Roy. 'Bayesian Learning for Safe High-Speed Navigation in Unknown Environments'. In: *ISRR*. 2015. DOI: 10.1007/978-3-319-60916-4_19 (cited on pages 24, 28).

[35] R. Bohlin and L. E. Kavraki. 'Path Planning Using Lazy PRM'. In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*. Vol. 1. IEEE. Apr. 2000, 521–528 vol.1. DOI: 10.1109/ROBOT.2000.844107 (cited on pages 24, 25).

[36] Aditya Mandalika et al. 'Generalized Lazy Search for Robot Motion Planning: Interleaving Search and Edge Evaluation via Event-Based Toggles'. In: *CoRR* abs/1904.02795 (2019) (cited on pages 24, 25).

[37] Butler W. Lampson. 'Lazy and Speculative Execution in Computer Systems'. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP '08. New York, NY, USA: Association for Computing Machinery, Sept. 20, 2008, pp. 1–2. DOI: 10.1145/1411204.1411205. (Visited on 03/18/2021) (cited on page 25).

[38] Evdokia Nikolova and David R. Karger. 'Route Planning under Uncertainty: The Canadian Traveller Problem'. In: *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*. AAAI'08. Chicago, Illinois: AAAI Press, July 13, 2008, pp. 969–974. (Visited on 03/12/2021) (cited on page 25).

[39] Z.W. Lim, D. Hsu, and W.S. Lee. 'Shortest Path under Uncertainty: Exploration versus Exploitation'. In: *Proc. Conf. on Uncertainty in Artificial Intelligence*. 2017 (cited on page 25).

[40] Brad Saund et al. 'The Blindfolded Robot: A Bayesian Approach to Planning with Contact Feedback'. In: *International Symposium on Robotics Research (ISRR)*. 2019 (cited on page 25).

[41] Zahy Bnaya, Ariel Felner, and Solomon Eyal Shimony. 'Canadian Traveler Problem with Remote Sensing'. In: *Twenty-First International Joint Conference on Artificial Intelligence*. 2009 (cited on page 25).

[42] S. Koenig and M. Likhachev. 'Fast Replanning for Navigation in Unknown Terrain'. In: *IEEE Transactions on Robotics* 21.3 (June 2005), pp. 354–363. DOI: 10.1109/TRO.2004.838026. (Visited on 03/16/2021) (cited on pages 25, 27).

[43] Sourabh Bhattacharya, Rafael Murrieta-Cid, and Seth Hutchinson. 'Optimal Paths for Landmark-Based Navigation by Differential-Drive Vehicles With Field-of-View Constraints'. In: *IEEE Transactions on Robotics* 23.1 (2007), pp. 47–59. DOI: 10.1109/TRO.2006.886841 (cited on page 26).

[44] Gabriele Costante et al. 'Perception-Aware Path Planning'. In: *CoRR* abs/1605.04151 (2016) (cited on page 26).

[45] N. Roy et al. 'Coastal Navigation-Mobile Robot Navigation with Uncertainty in Dynamic Environments'. In: *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*. Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C). Vol. 1. May 1999, 35–40 vol.1. DOI: 10.1109/ROBOT.1999.769927 (cited on page 26).

[46] Nicholas Roy and Sebastian Thrun. 'Coastal Navigation with Mobile Robots'. In: *Proceedings of the 12th International Conference on Neural Information Processing Systems*. NIPS'99. Cambridge, MA, USA: MIT Press, Nov. 29, 1999, pp. 1043–1049 (cited on page 26).

[47] J. P. Gonzalez and A. Stentz. 'Planning with Uncertainty in Position an Optimal and Efficient Planner'. In: *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems. Aug. 2005, pp. 2435–2442. DOI: 10.1109/IROS.2005.1545048 (cited on page 26).

[48] R. Platt et al. 'Belief Space Planning Assuming Maximum Likelihood Observations'. In: Robotics: Science and Systems VI. Vol. 06. June 27, 2010. (Visited on 03/18/2021) (cited on page 26).

[49] Jur van den Berg, Sachin Patil, and Ron Alterovitz. 'Motion Planning under Uncertainty Using Iterative Local Optimization in Belief Space'. In: *The International Journal of Robotics Research* 31.11 (Sept. 1, 2012), pp. 1263–1278. DOI: 10.1177/0278364912456319. (Visited on 03/18/2021) (cited on page 26).

[50] Ali-akbar Agha-mohammadi et al. 'SLAP: Simultaneous Localization and Planning Under Uncertainty via Dynamic Replanning in Belief Space'. In: *IEEE Transactions on Robotics* 34.5 (Oct. 2018), pp. 1195–1214. DOI: 10.1109/TRO.2018.2838556 (cited on page 26).

[51] Ruben Martinez-Cantin et al. 'A Bayesian Exploration-Exploitation Approach for Optimal Online Sensing and Planning with a Visually Guided Mobile Robot'. In: *Autonomous Robots* 27.2 (Aug. 1, 2009), pp. 93–103. DOI: 10.1007/s10514-009-9130-2. (Visited on 03/19/2021) (cited on page 26).

[52] Vadim Indelman, Luca Carlone, and Frank Dellaert. 'Planning in the Continuous Domain: A Generalized Belief Space Approach for Autonomous Navigation in Unknown Environments'. In: *The International Journal of Robotics Research* 34.7 (June 1, 2015), pp. 849–882. DOI: 10.1177/0278364914561102. (Visited on 03/12/2021) (cited on page 26).

[53] L. Carlone and D. Lyons. 'Uncertainty-Constrained Robot Exploration: A Mixed-Integer Linear Programming Approach'. In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. 2014 IEEE International Conference on Robotics and Automation (ICRA). May 2014, pp. 1140–1147. DOI: 10.1109/ICRA.2014.6906997 (cited on page 26).

[54] Ali-akbar Agha-mohammadi. *SMAP: Simultaneous Mapping and Planning on Occupancy Grids*. Sept. 18, 2016. URL: http://arxiv.org/abs/1608.04712 (visited on 12/04/2021) (cited on page 26).

[55] Jin Bao, Wang Shuguo, and Yili Fu. 'Sensor-Based Motion Planning for Robot Manipulators in Unknown Environments'. In: *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems. Aug. 2005, pp. 199–204. DOI: 10.1109/IROS.2005.1545512 (cited on pages 27, 73).

[56] D. Fox, W. Burgard, and S. Thrun. 'The Dynamic Window Approach to Collision Avoidance'. In: *IEEE Robotics Automation Magazine* 4.1 (Mar. 1997), pp. 23–33. DOI: 10.1109/100.580977 (cited on page 27).

[57] Sara Bouraine, Thierry Fraichard, and Hassen Salhi. 'Provably Safe Navigation for Mobile Robots with Limited Field-of-Views in Dynamic Environments'. In: *Autonomous Robots* 32.3 (Apr. 1, 2012), pp. 267–283. DOI: 10.1007/s10514-011-9258-8. (Visited on 03/12/2021) (cited on page 28).

[58] Andreas Krause and Daniel Golovin. 'Submodular Function Maximization'. In: *Tractability*. Ed. by Lucas Bordeaux et al. Cambridge: Cambridge University Press, 2013, pp. 71–104. DOI: 10.1017/CBO9781139177801.004. (Visited on 01/03/2022) (cited on page 30).

[59] Sven Koenig and Maxim Likhachev. 'D* Lite'. In: *Aaai/iaai* 15 (2002) (cited on page 30).

[60] Bhaskara Marthi. 'Robust Navigation Execution by Planning in Belief Space'. In: *Robotics: Science and Systems*. MIT Press, 2012. DOI: 10.15607/RSS.2012.VIII.037 (cited on page 30).

[61] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Fourth edition. Pearson Series in Artificial Intelligence. Hoboken: Pearson, 2021 (cited on pages 32, 33).

[62] Neil J. A. Sloane and The OEIS Foundation Inc. *The On-Line Encyclopedia of Integer Sequences*. URL: http://oeis.org/?language=english (cited on page 32).

[63] Neil J. A. Sloane and The OEIS Foundation Inc. *The On-Line Encyclopedia of Integer Sequences, A064298, Square Array Read by Antidiagonals of Self-Avoiding Rook Paths Joining Opposite Corners of n X k Board*. URL: https://oeis.org/A064298 (cited on page 34).

[64] Luke Shimanuki and Brian Axelrod. 'Hardness of 3D Motion Planning under Obstacle Uncertainty'. In: *WAFR*. 2018 (cited on page 40).

[65] L. E. Kavraki et al. 'Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces'. In: *IEEE Transactions on Robotics and Automation* 12.4 (Aug. 1996), pp. 566–580. DOI: 10.1109/70.508439 (cited on page 43).

[66] nLab authors. *nLab: Biased Definition*. URL: https://ncatlab.org/nlab/show/biased+definition (visited on 11/09/2021) (cited on page 45).

[67] Samuel Prentice and Nicholas Roy. 'The Belief Roadmap: Efficient Planning in Belief Space by Factoring the Covariance'. In: *The International Journal of Robotics Research* 28.11-12 (Nov. 1, 2009), pp. 1448–1465. DOI: 10.1177/0278364909341659. (Visited on 03/12/2021) (cited on page 47).

[68] Gustavo Goretkin, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. 'Look Before You Sweep: Visibility-Aware Motion Planning'. In: *Algorithmic Foundations of Robotics XIII*. Ed. by Marco Morales et al. Springer Proceedings in Advanced Robotics. Cham: Springer International Publishing, 2020, pp. 373–388. DOI: 10.1007/978-3-030-44051-0_22 (cited on pages 50, 59, 73, 77, 80, 84, 85).

[69] Tomás Lozano-Pérez, Matthew T. Mason, and Russell H. Taylor. 'Automatic Synthesis of Fine-Motion Strategies for Robots'. In: *The International Journal of Robotics Research* 3.1 (Mar. 1, 1984), pp. 3–24. DOI: 10.1177/027836498400300101. (Visited on 11/29/2021) (cited on page 53).

[70] John H. Reif. 'Complexity of the Mover's Problem and Generalizations'. In: *20th Annual Symposium on Foundations of Computer Science (Sfcs 1979)*. 20th Annual Symposium on Foundations of Computer Science (Sfcs 1979). Oct. 1979, pp. 421–427. DOI: 10.1109/SFCS.1979.10 (cited on page 62).

[71] John Canny and John Reif. 'New Lower Bound Techniques for Robot Motion Planning Problems'. In: *28th Annual Symposium on Foundations of Computer Science (Sfcs 1987)*. 28th Annual Symposium on Foundations of Computer Science (Sfcs 1987). Oct. 1987, pp. 49–60. DOI: 10.1109/SFCS.1987.42 (cited on page 62).

[72] Lawrence H. Erickson and Steven M. LaVall. 'A Simple, but NP-hard, Motion Planning Problem'. In: *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*. AAAI'13. Bellevue, Washington: AAAI Press, July 14, 2013, pp. 1388–1393 (cited on pages 62, 63).

[73] Luke Shimanuki and Brian Axelrod. 'Hardness of Motion Planning with Obstacle Uncertainty in Two Dimensions'. In: *The International Journal of Robotics Research* 40.10-11 (Sept. 1, 2021), pp. 1151–1166. DOI: 10.1177/0278364921992787. (Visited on 01/03/2022) (cited on pages 62, 63).

[74] Leonidas J. Guibas et al. 'Visibility-Based Pursuit-Evasion in a Polygonal Environment'. In: *Algorithms and Data Structures*. Ed. by Frank Dehne et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1997, pp. 17–30. DOI: 10.1007/3-540-63307-3_45 (cited on page 63).

[75] Hubert Nguyen and NVIDIA Corporation. *GPU Gems 3*. Upper Saddle River, N.J.: Addison-Wesley, 2008 (cited on page 68).

[76] Steve Macenski, David Tsai, and Max Feinberg. 'Spatio-Temporal Voxel Layer: A View on Robot Perception for the Dynamic World'. In: *International Journal of Advanced Robotic Systems* 17.2 (Mar. 1, 2020), p. 1729881420910530. DOI: 10.1177/1729881420910530. (Visited on 10/20/2021) (cited on page 70).

[77] Sergiy Bogomolov et al. 'JuliaReach: A Toolbox for Set-Based Reachability'. In: *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*. 2019, pp. 39–44 (cited on page 70).

[78] George W Hart. *Multidimensional Analysis: Algebras and Systems for Science and Engineering*. New York, NY: Springer New York, 1995. (Visited on 11/29/2021) (cited on page 71).

[79] Advait Jain et al. 'Reaching in Clutter with Whole-Arm Tactile Sensing'. In: *The International Journal of Robotics Research* 32.4 (Apr. 1, 2013), pp. 458–482. DOI: 10.1177/0278364912471865. (Visited on 09/29/2021) (cited on page 73).

[80] Anna Yershova and Steven M. LaValle. 'Improving Motion-Planning Algorithms by Efficient Nearest-Neighbor Searching'. In: *IEEE Transactions on Robotics* 23.1 (Feb. 2007), pp. 151–157. DOI: 10.1109/TRO.2006.886840 (cited on page 76).

[81] Christer Ericson. *Real-Time Collision Detection*. CRC Press, Dec. 22, 2004. 633 pp. (cited on page 76).

[82] James R. Driscoll et al. 'Making Data Structures Persistent'. In: *Journal of Computer and System Sciences* 38.1 (Feb. 1, 1989), pp. 86–124. DOI: 10.1016/0022-0000(89)90034-2. (Visited on 09/03/2021) (cited on page 77).

[83] Jon Louis Bentley. 'Multidimensional Binary Search Trees Used for Associative Searching'. In: *Communications of the ACM* 18.9 (Sept. 1, 1975), pp. 509–517. DOI: 10.1145/361002.361007. (Visited on 09/03/2021) (cited on pages 77, 83).

[84] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. 'Planning and Acting in Partially Observable Stochastic Domains'. In: *Artificial Intelligence* 101.1 (May 1, 1998), pp. 99–134. DOI: 10.1016/S0004-3702(98)00023-X. (Visited on 09/03/2021) (cited on page 79).

[85] Lawson L. S. Wong, Leslie Pack Kaelbling, and Tomas Lozano-Perez. 'Not Seeing Is Also Believing: Combining Object and Metric Spatial Information'. In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. 2014 IEEE International Conference on Robotics and Automation (ICRA). May 2014, pp. 1253–1260. DOI: 10.1109/ICRA.2014.6907014 (cited on page 79).

[86] R. Platt et al. 'Belief Space Planning Assuming Maximum Likelihood Observations'. In: Robotics: Science and Systems VI. Vol. 06. June 27, 2010. (Visited on 09/03/2021) (cited on page 80).

[87] Jon Louis Bentley and James B Saxe. 'Decomposable Searching Problems I. Static-to-dynamic Transformation'. In: *Journal of Algorithms* 1.4 (Dec. 1, 1980), pp. 301–358. DOI: 10.1016/0196-6774(80)90015-2. (Visited on 09/03/2021) (cited on page 81).

[88] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Series in Computer Science. Reading, Mass: Addison-Wesley, 1990. 493 pp. (cited on page 81).

[89] Jeff Bezanson et al. 'Julia: A Fresh Approach to Numerical Computing'. In: *SIAM Review* 59.1 (Jan. 1, 2017), pp. 65–98. DOI: 10.1137/141000671. (Visited on 09/09/2021) (cited on page 87).

[90] *Julia Base Dict Implementation Resizing julia/dict.jl · JuliaLang/Julia*. URL: https://github.com/JuliaLang/julia/blob/fd7bc03e1dc3fe2d45957d41944f32a7a20e08bf/base/dict.jl#L354-L370 (visited on 09/14/2021) (cited on page 89).

[91] Haim Kaplan. 'Persistent Data Structures *'. In: *Handbook of Data Structures and Applications*. 2nd ed. Chapman and Hall/CRC, 2017 (cited on page 89).

[92] Chris Okasaki. 'Purely Functional Data Structures'. Princeton University, 1996. 162 pp. (cited on page 89).

[93] Antonin Guttman. 'R-Trees: A Dynamic Index Structure for Spatial Searching'. In: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. SIGMOD '84. New York, NY, USA: Association for Computing Machinery, June 1, 1984, pp. 47–57. DOI: 10.1145/602259.602266. (Visited on 09/02/2021) (cited on page 90).