Article

The International Journal of Robotics Research
32(11) 1342–1357
© The Author(s) 2013
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/0278364913488428
ijr.sagepub.com

Planning for provably reliable navigation using an unreliable, nearly sensorless robot

Jeremy S. Lewis and Jason M. O'Kane

Abstract

This paper addresses a navigation problem for a certain type of simple mobile robot modeled as a point moving in the plane. The only requirement on the robot is that it must be able to translate in a desired direction, with bounded angular error (measured in a global reference frame), until it reaches the nearest obstacle in its motion direction. One straightforward realization of this capability might use a noisy compass and a contact sensor. We present a planning algorithm that enables such a robot to navigate reliably through its environment. The algorithm constructs a directed graph in which each node is labeled with a subset of the environment boundary. Each edge of the graph is labeled with a sequence of actions that can move the robot from any location in one such set to some location in the other set. We use a variety of local planners to generate the edges, including a "corner-finding" technique that allows the robot to travel to and localize itself at a convex vertex of the environment boundary. The algorithm forms complete plans by searching the resulting graph. We have implemented this algorithm and present results from both simulation and a proof-of-concept physical realization.

Keywords

Planning, navigation

1. Introduction

The ability to navigate reliably through a cluttered environment is a fundamental capability for mobile robots. Navigation can be a challenging problem because of the coupled difficulties of finding a path from the robot's starting location to its goal and of executing such a path successfully in spite of unpredictable actuation and limited sensing. Typical navigation methods take a decoupled approach, in which *path selection* and *path execution* are handled separately. The former phase chooses a path for the robot to follow without considering sensing issues, and the latter uses the robot's sensors to execute the chosen path. The primary limitation of this approach is that it is unsuitable for situations in which the robot must choose its path, or portions thereof, specifically to reduce or eliminate uncertainty.

In this paper, we present a unified approach that considers sensing and movement uncertainty directly in the process of path selection. Our approach has parallels to prior work on coastal navigation (Roy and Thrun, 1999), but applies in a *minimalist* setting, considering a robot whose only capability is to translate in a desired direction, subject to bounded angular error, until reaching the nearest obstacle in its direction of motion. This model could be realized, for example, using a robot with no sensors other than a noisy compass and a binary contact sensor. Our planner directly considers the robot's uncertainty to generate plans containing some steps that move the robot toward the goal, and other steps intended specifically to reduce uncertainty when necessary. Figure 1 (see also Extension 1) shows a simulated execution of a plan generated by our algorithm.

In addition to the underlying theoretical goal—namely, understanding the complexity of robotics problems such as navigation by studying the information required by the robot to solve them—there are also several practical reasons that planning with minimalistic sensing can be interesting. Most obviously, knowing how robotics problems can be solved with very simple robots can lead to inexpensive hardware designs. This consideration is particularly important for deployments that require multiple cooperating robots. Second, knowledge of how to control robots in spite of sensing limitations can be used to enhance the robustness of sensor-rich robots by providing guidance on

Department of Computer Science and Engineering, University of South Carolina, Columbia, SC, USA

Corresponding author:
Jason M. O'Kane, Department of Computer Science and Engineering, University of South Carolina, 315 Main Street, Swearingen Engineering Center, Columbia, SC 29208, USA.
Email: jokane@cse.sc.edu

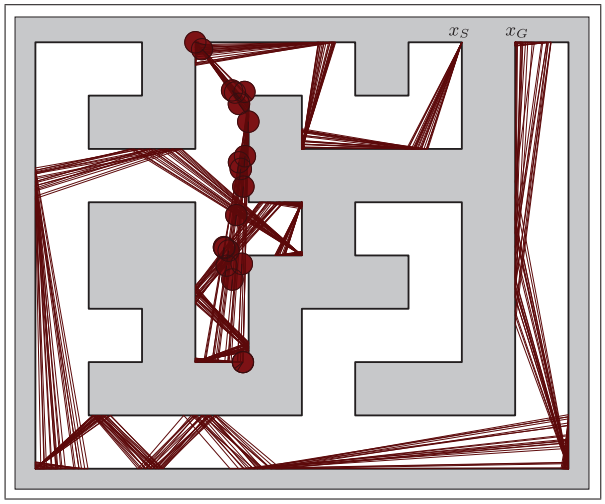

Fig. 1. A plan generated by our algorithm to navigate from a start state x_S to a goal state x_G. The figure shows a snapshot of 20 simultaneous simulations of a robot executing that plan, each experiencing sensor noise of up to 10° in either direction at each step. Each filled circle denotes the position of the robot in one of these simulations at the time of the snapshot. In spite of the sensor noise, the robot is guaranteed to reach its goal. Note that the robot uses several of the environment corners to relocalize itself during its journey.

how to proceed when sensor systems fail. This can be particularly valuable for scenarios in which the deployment cost is very high. Third, by identifying very small amounts of information that are most helpful for completing a task, we can select less complex, more reliable sensors that collect only this helpful information. Note that this approach is complementary but distinct from the ongoing research on robot swarms (Kloetzer and Belta, 2007; McLurkin et al., 2012; Mier-y-Teran-Romero et al., 2012): we consider robot hardware that is well-suited for swarm methods, but equip it with software that explicitly represents uncertainty and forms plans for the future, rather than relying on the emergence of desirable behavior from the interactions of simple control rules.

Our robot's goal is to navigate between given start and goal locations in a known environment. The intuition of our approach is that the robot is able—using some combination of physical sensors and actuators—to translate along a commanded direction within known bounds on angular error, until it reaches an obstacle. When this motion completes, the robot will not necessarily know its own position, but it *will* be able to identify a portion of the environment boundary as a set of *possible states*. The navigation planning process proceeds by constructing and searching a graph whose nodes are labeled with these sets of possible states. Each edge between these nodes is labeled with a sequence of actions that the robot can execute to ensure movement from the set of possible states represented by one node to the set of possible states represented by another. We present a family of local planners that generate these edges and the short action sequences that label them.

Using the graph, the planning process is simply a question of constructing a sequence of directed edges that connect nodes corresponding to the initial and goal states. To ensure that this process can be completed efficiently, we use a priority function to order the node pairs considered for connection, based on an estimate of their applicability to the overall plan and by their potential for connection by one of our local planners.

One noteworthy aspect of this approach is that it generates open-loop strategies: the output of our algorithm is a sequence of actions for the robot to execute, without any branching. The robot does not receive any feedback during its execution about its progress to the goal. This feature, which arises directly from the extreme limitations of the robot's sensors, means that the planning algorithm must take responsibility to generate plans that guarantee that the robot will reach the goal, in spite of the lack of feedback.

The specific contributions of this paper are as follows. (1) After reviewing related work in Section 2, we introduce, in Section 3, a navigation problem for a robot with a map that can translate in a desired direction, with bounded angular error, until it reaches the nearest obstacle in its direction of motion. (2) We describe, in Section 4, a planning algorithm for this problem and present, in Section 5, an implementation of this algorithm and evaluate the algorithm's performance both in simulation and in hardware experiments. (3) In Section 6, we discuss the limitations of our algorithm and suggest avenues for potential future work.

Preliminary versions of this work appeared at ICRA 2010 (Lewis and O'Kane, 2010) and ICRA 2012 (Lewis and O'Kane, 2012). This version is substantially rewritten and presents new theoretical and experimental results.

2. Related work

This section reviews some of the existing related research.

2.1. Planning under sensing limitations

In a general sense, our planning algorithm builds upon the idea of pre-image backchaining introduced by Lozano-Pérez et al. (1984). Their research describes the notion of a fine-motion strategy as an effective counter to position uncertainty in compliant motions. Our approach is similar in that we consider an error cone—that is, a range of possible uncertainty values for each translation made by our robot—that expands the set of possible states from a single known state to some larger set of states derived from a known bound on error.

There is also a strong influence from the idea of landmark-based navigation proposed by Lazanas and Latombe (1992). They suggest the use of landmarks such that, while the robot is in proximity to a landmark, the robot is able to execute error-free actions. They also assert

that the robot is able to recognize when it has achieved its goal. In contrast, our robot has no goal-detecting sensor, nor does our planner depend on the robot explicitly sensing that it has achieved its goal state. Our algorithm also never assumes error-free actions by the robot nor an exact knowledge of any state after leaving the initial state. Instead, we exploit the geometric features of the environment as "implicit landmarks" to reduce uncertainty *without* explicit sensing.

Our approach is also inspired by Erdmann and Mason's (1988) work on sensorless manipulation. Our work follows suit with an inspection of the robot's environment, rather than any specific assumptions about the environment as in Lozano-Pérez et al. (1984). Our synthesis of these works results in a planner that uses parts of the environment as landmarks, by describing a careful iterative motion process to eliminate uncertainty periodically throughout the robot's execution. By determining landmarks from plentiful environment features, in this case, convex vertices, we show that a very simple robot is able to solve problems previously considered only for robots with more powerful sensors.

Our work considers the uncertainty of the robot as a guide in path planning. This approach is similar to planning using coastal navigation techniques (Roy and Thrun, 1999), which moves from one landmark to another using range sensors to detect landmarks. More recent work on belief roadmaps (Prentice and Roy, 2009), feedback-based information roadmaps (Agha-mohammadi et al., 2012), and randomized belief-space trees (Hauser, 2010) has integrated a consideration of uncertainty into the path planning process. Our model is unique because of the scarcity of information available to our robot.

2.2. Minimalism

The goal of simplifying sensing and actuation in robotic systems, while retaining, of course, the capability to solve meaningful problems, is not a new idea. Many tasks are considered with this approach including manipulation in general (Lozano-Pérez et al., 1984; Erdmann, 1986, 1995; Akella and Mason, 1998), part orientation specifically (Erdmann and Mason, 1988; Goldberg, 1993; Akella et al., 2000; van der Stappen et al., 2000; Berretty et al., 2001; Moll and Erdmann, 2002), navigation (Lazanas and Latombe, 1992; Lumelsky and Tiwari, 1994; Blum et al., 1997; Kamon and Rivlin, 1997; Deng et al., 1998; Kamon et al., 1999), and mapping (Ó. Dúnlaing and Yap, 1982; Choset and Burdick, 2000b,a; Acar and Choset, 2001; Tovar et al., 2004; Katsev et al., 2011).

In addition, the much more general question of how to determine the minimal sensing required to complete a given task has been considered in various ways (Blum and Kozen, 1978; Donald, 1995; Erdmann, 1995). This methodology of minimalist robotics research can arguably be traced back to Whitney (1986), and was clearly articulated by Mason (1993). Often the idea behind such an approach is that, by studying how a problem can be solved by a robot with weaker sensing and motion capabilities, a greater understanding of the problem itself can be obtained. This notion of some robots being weaker than others has been formalized in terms of one robot's ability to "simulate" another in a precisely defined way (O'Kane and LaValle, 2008).

2.3. Localization with limited sensing

More specifically, a few of the second author's prior papers have considered similar robot models for localization problems. O'Kane and LaValle (2007) showed that the global active localization problem can be solved by an idealized robot with a precise compass and a binary contact sensor. In the current paper, we consider a more realistic model for robot motion that includes substantial errors, and show that many navigation problems can still be solved under this model.

Erickson et al. (2008) extended those localization results using a probabilistic model for sensor noise. That research also uses the idea of an error cone to model uncertainty in rotation, which moves a robot's belief state from one probability distribution to another. The differences in our work are that we are solving a navigation problem, that we use non-deterministic reasoning to achieve guarantees of success, and that we use exact geometry rather than a coarse discretization of the environment.

3. Problem statement

This section formalizes the navigation problem we consider.

A point robot moves in a closed, bounded, polygonal region $W \subset \mathbb{R}^2$ of the plane.[1] The robot has a complete and accurate map of its environment. A vertex v of W is *convex* if the neighborhood of v in W is convex. Formally, let $B(v, \epsilon)$ denote the open ball with radius ϵ centered at v. A vertex v is defined as convex if there exists some $\epsilon > 0$ such that $B(v, \epsilon) \cap W$ is a convex set. Informally, note that convex vertices are formed whenever the two incident edges of a vertex form an interior angle less than or equal to π radians.

Our model for the motions of this robot has the following elements:

1. The *state space* $X = W$ is simply the robot's environment.
2. The *action space* $U = [0, 2\pi]/\sim$ is the set of planar angles, in which $\sim$ is an equivalence relation under which the endpoints 0 and 2π are identified. To execute an action $u \in U$, the robot moves in direction u, subject to the error described below, until it reaches the environment boundary.
3. Time proceeds in a series of *stages*, numbered $k = 1, 2, \ldots$. In each stage, the robot chooses and executes a single action. At stage k, the robot's state is denoted by x_k and its action is denoted by u_k.

4. Rotation errors are modeled as additive interference by an imaginary adversary called *nature*. In each stage, nature chooses a *nature action* $\theta_k \in \Theta$. Nature's action space $\Theta = (-\theta_{\max}, +\theta_{\max})$ is an interval of possible error values. Note that because we are interested in worst-case guarantees of success, we need not consider any probabilities over Θ; nature is free to choose any $\theta_k \in \Theta$ at each stage, without being constrained by any stochastic model. The robot has no knowledge of nature's choice, nor any way to observe it directly or indirectly.
5. The *state transition function* $f : X \times U \times \Theta \to X$ describes how the state changes in response to the robot's and nature's actions. The current state x_k, combined with the robot's action u_k and nature's action θ_k, determines the next state x_{k+1}, so that $x_{k+1} = f(x_k, u_k, \theta_k)$. Specifically, $f(x_k, u_k, \theta_k)$ is defined as the opposite endpoint of the longest segment in X, starting at x_k and moving in direction $u_k + \theta_k$. Note that, due to the influence of nature, the robot does not know x_{k+1} exactly.

   For convenience, we occasionally abuse this notation to apply multiple stages' worth of actions at once, so that

$$x_{k+i} = f(x_k, u_k, \theta_k, u_{k+1}, \theta_{k+1}, \ldots, u_{k+i}, \theta_{k+i}).$$

Note that any physical robot that instantiates this transition model is a suitable platform for our algorithm. This includes, for example, rotating robots equipped with compasses and binary contact sensors, omnidirectional robots that can move in the desired direction without rotating, and robots that lack a dedicated compass sensor, but can recover the same information using contextual information (cf. Section 5.2).

The robot's goal, given W and $\theta_{\max}$, along with starting and goal states $x_S, x_G \in W$ and an accuracy bound δ, is to choose a sequence of actions $u_1, \ldots, u_K$ such that

$$\|x_G - f(x_S, u_1, \theta_1, \ldots, u_K, \theta_K)\| < \delta \qquad (1)$$

for any nature action sequence $\theta_1, \ldots, \theta_K \in \Theta$. That is, we seek actions that drive the robot from x_S to a point close to x_G, regardless of nature's actions. The accuracy bound δ is needed because the robot's motion error and sensor limitations prevent it from ever knowing with certainty that is has reached x_G exactly.

4. Algorithm description

This section describes an algorithm for the navigation problem introduced in Section 3.

4.1. Algorithm overview

The intuition of the algorithm is to construct a directed graph that describes the navigability of the environment. Each node of this graph represents a set of states—sometimes a singleton—in which the robot may, at some point during its execution, know that it lies. We say that the robot is "at node v" when the robot knows, based on its history of actions, that its (unknown) true state must be somewhere within the set of states represented by v. We use two classes of nodes: *point nodes* representing convex vertices of W, and *segment nodes* representing positional uncertainty along some boundary edge of W. The precise technique we use to generate these nodes appears in Section 4.3.

Each edge in the graph is labeled with a sequence of actions $u_i, \ldots, u_{i+K}$. The interpretation is that, if nodes v and v' are connected by an edge, then starting from any state in the state set corresponding to v, and under any sequence of nature actions, if the robot executes the edges' actions it will reach some final state in the set corresponding to v'. Each edge, therefore, describes how the robot can travel reliably between two of the graph's nodes. The specific local planners we use to generate these action sequences are described in Section 4.4.

Finally, to avoid exhaustively constructing the entire graph, we propose a priority scheme (Section 4.5) to focus the algorithm's computation on attempting to generate edges between node pairs for which both (1) the existence of an edge would be helpful for forming a complete plan, and (2) it appears promising that our local planners will be able to generate such an edge.

A description of the graph construction process we use for efficiently generating a complete plan appears in Section 4.6.

4.2. Notation and conventions

Throughout this section, we use the following conventions.

4.2.1. Boundary segments A segment S along the boundary of W is identified by its endpoints, which we call the *source point* $\mathrm{src}(S)$ and the *target point* $\mathrm{tar}(S)$. By convention, we choose the source point and target point of any boundary segment S so that the free space of W is on the counterclockwise side of the vector $\mathrm{tar}(S) - \mathrm{src}(S)$. A segment S is *degenerate* if $\mathrm{src}(S) = \mathrm{tar}(S)$. In this case, the segment is a single point.

4.2.2. Error cones and safe actions Given a segment $S \subseteq W$, an action u, and an error bound $\theta_{\max}$, the *error cone* $e(S, u, \theta_{\max})$ is defined as the portion of W through which a robot could potentially pass when executing u from any state in S. Figure 2 shows an example error cone.

We call an action u a *safe action* from a segment S if the far boundary of $e(S, u, \theta_{\max})$ is contained within a single boundary edge of W. An interval of actions $(u_1, u_2) \subset U$ is an *interval of safe actions* from S if every action in the interior of the interval is safe from S.

The notion of safe actions is crucial to our algorithm, because it captures the notion of actions that the robot can execute without losing track of which environment edge it is

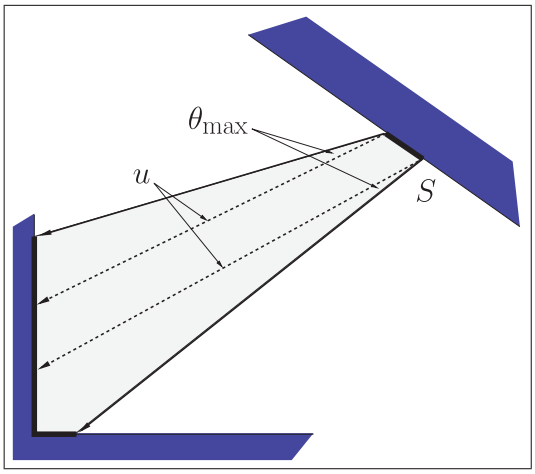

Fig. 2. An error cone (shaded region). Because the far boundary spans multiple edges of W, the illustrated action u is not a safe action.

touching. Our algorithm generates plans consisting only of safe actions. The idea of safe actions plays a role in implementing the algorithm on a real robot (see Section 5.2), and also has a strong impact on the completeness properties of any algorithm for this problem (see Section 6).

4.2.3. Ray shooting Several parts of our algorithm utilize a function called SHOOTRAY. The inputs to this function are the environment W, a query point $q \in W$, and a planar direction. The SHOOTRAY function returns the first boundary point contacted by a point starting at p and moving through W in the given direction. This ray-shooting operation is well-known in computational geometry. After preprocessing W in $O(n)$ time, each query takes $O(\log n)$ time, in which n is the number of vertices in W (Szirmay-Kalos and Marton, 1998).

4.3. Graph nodes

The first step of the algorithm is to generate a collection of graph nodes. We use two distinct classes of nodes.

4.3.1. Point nodes First, we create one *point node*, consisting of a degenerate segment, at each convex vertex of the boundary of W. For each point node v in the graph, we write $p(v)$ to denote the environment point associated with v.

The convex vertices are useful choices because of the combination of two factors. First, as singleton points, they represent situations at which the robot is fully localized. Therefore, it is often desirable to reach these nodes. Second, as we show in Sections 4.4.3 and 4.4.4, there often exists a sequence of actions that *does* reach these kinds of nodes under certain conditions. As such these desirable nodes are often reachable.

We also create two additional point nodes v_S and v_G, at the start and goal positions (if they are not already convex vertices), so that $p(v_S) = x_S$ and $p(v_G) = x_G$.

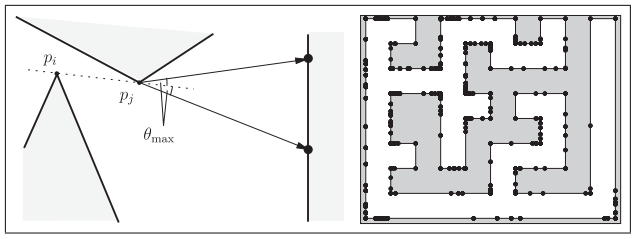

Fig. 3. Left: Two delimiting points (right side) generated by a pair of mutually visible vertices. Right: All of the delimiting points generated by Algorithm 1 with $\theta_{\max} = 10^\circ$ in a sample environment.

4.3.2. Segment nodes The second class of nodes we include in our graph are *segment nodes*. Each segment node corresponds to a non-degenerate segment along a single edge of the boundary of W. The intuition is that these nodes allow the robot to tolerate some temporary uncertainty that will eventually be resolved by visiting a point node. For each segment node v, we write $S(v)$ to denote the segment corresponding to v, and (in a slight abuse of notation), src(v) and tar(v) to refer to the endpoints src($S(v)$) and tar($S(v)$) of that segment.

There are, of course, uncountably many potential segment nodes, and no efficient algorithm is able to consider all of these explicitly as part of the graph. Which segment nodes should be included in the graph? Our approach uses *mutually visible vertices* to select nodes, accounting for the uncertainty of the system. The intuition is that, to move safely past an obstacle vertex, the robot cannot aim directly in the direction of the vertex, but instead must aim away by an angle at least $\theta_{\max}$.

We consider all pairs of mutually visible vertices $p_i, p_j \in W$. For each pair, we shoot four rays, two of which extend from p_i and two of which extend from p_j. The rays travel in the directions angle$(p_j - p_i) \pm \theta_{\max}$ and angle$(p_i - p_j) \pm \theta_{\max}$, respectively, until they contact the environment boundary. The resulting points of contact, along with the endpoints of each environment edge, are called *delimiting points*. Figure 3 shows the rays originating at an example p_j, along with all of the delimiting points in an example environment.

Using the delimiting points, we generate, for each environment edge, one segment node for each ordered pair of distinct delimiting points. The details of the process appear in Algorithm 1.

To quantify the number of nodes generated by Algorithm 1, let n denote the number of vertices of W and let l_i denote the number of delimiting points on edge i. For any two vertices $p_i \neq p_j$, the algorithm will create at most two delimiting points originating from p_j, each of which lies on a single edge. Therefore, we know that $\sum_{i=1}^{n} l_i < 2n^2$. Note also that Algorithm 1 creates $\left(l_i^2 + l_i\right)/2$ nodes for edge i. Therefore (recalling that squaring is a convex and therefore superadditive function), the total number of nodes created is $\sum_{i=1}^{n} \left[\left(l_i^2 + l_i\right)/2\right] \leq \sum_{i=1}^{n} \left[\left(2l_i^2\right)/2\right] = \sum_{i=1}^{n} \left(l_i^2\right) \leq$

Algorithm 1 GENERATESEGMENTNODES($W, \theta_{\max}$)

```
D ← empty set of delimiting points
for all vertices p_i ∈ W do
    for all vertices p_j ∈ W do
        if p_i ≠ p_j and p_j is not a convex vertex and p_j is
        visible from p_i in W then
            ψ ← angle(p_j − p_i)
            D ← D ∪ {SHOOTRAY(p_i, ψ + θ_max, W),
            SHOOTRAY(p_i, ψ − θ_max, W)}
        end if
    end for
end for
Q ← empty set of segments
for all boundary edges e ∈ W do
    D′ ← D ∩ e
    D′ ← D′ ∪ {src(e), tar(e)}
    SORT D′ by the distance from src(e)
    for p ← 1 ← count(D′) do
        for q ← p + 1 ← count(D′) do
            Q ← Q ∪ {segment between D′[p] and D′[q]}
        end for
    end for
end for
return Q
```

$\left(\sum_{i=1}^{n} l_i\right)^2 \leq (2n^2)^2 \in O(n^4)$. Note that, in choosing a finite set of segment nodes, we leave open the possibility that a solution may exist for the original navigation problem, but that no solution exists using only that set of segment nodes. Our experimental results (see Section 5) show that this set of segment nodes is generally very effective. Section 6 discusses completeness issues in greater detail.

4.4. Graph edges

Given this set of graph nodes, we now turn our attention to the question of when the robot can navigate between a given pair of those nodes. Recall that a directed edge in our graph $v_i \longrightarrow v_j$ represents the existence of a known sequence of actions which reliably brings the robot from v_i to v_j. In this section, we present a collection of local planners that we use to generate such edges.

We first consider a few relatively simple planners that attempt direct, single step connections from a point node to a segment node (Section 4.4.1); and from one segment node to another (Section 4.4.2). However, the true "heart" of the algorithm is a pair of local planners, described in Sections 4.4.3 and 4.4.4, that generate edges incoming to point nodes. Such transitions are crucial to our planner because they enable the robot to re-localize itself along the way toward its goal. Finally, we consider a point-to-point local planner specifically designed to navigate "long hallway" structures that are problematic for the other local planners (Section 4.4.5).

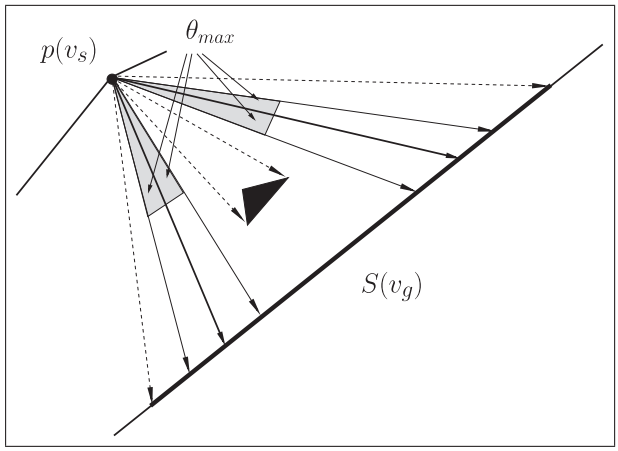

Fig. 4. Testing for a single-action edge from a point node v_s to a segment node v_g. The dashed rays represent the events of the radial sweep performed by Algorithm 2. In this case, there are two distinct intervals of safe actions. Therefore, Algorithm 3 would successfully generate an edge from v_s to v_g. The thick solid rays show two choices for the action with which this edge could be labeled.

4.4.1. Point node to segment node Let us first consider the most straightforward local planner, which attempts to connect from point nodes to segment nodes. To generate an edge that moves from a point node v_s to a segment node v_g in a single step, we need only to identify the intervals of motion directions which, when carried out starting at $p(v_s)$, will reach the segment $S(v_g)$ without encountering any other obstacles along the way. From that set we then determine whether any interval of motion directions spans at least $2\theta_{\max}$. If so, we generate an edge labeled with a single action, selected from that interval to have at least $\theta_{\max}$ clearance on either side. Said another way, if we can fit an entire error cone from the starting point onto the goal segment, then we generate an edge that makes that transition.

To search for such an interval of safe actions, our algorithm performs a radial sweep about $p(v_s)$, testing between each pair of consecutive obstacle vertices reached by the sweep ray. Note that, because there are no other environment vertices between each consecutive pair of obstacle vertices, a single ray shooting query is sufficient to test whether the entire range is safe. Figure 4 shows an example in which there are two intervals of safe actions. Algorithm 2 shows the details of how we compute these intervals of safe actions using the radial sweep; Algorithm 3 shows the local planner itself. (These two parts of the planner are shown separately because we reuse Algorithm 2 later.) The algorithm's run time is dominated by the sorting of environment vertices, and therefore is in $O(n \log n)$, in which n is the number of environment vertices.

4.4.2. Segment node to segment node Our second local planner attempts to generate edges from one segment node

Algorithm 2 SAFEACTIONS_POINTTOSEGMENT $(W, v_s, v_g, \theta_{\max})$

```
A ← empty list of angles
for all vertices v ∈ W do
    A.insert( angle( v − p( v_s) ) )
end for
sort A into clockwise order about p( v_s)
U_p ← empty set of actions
for all i ∈ {1, . . . , count( A) } do
    if 2θ_max < angle( A[i].next − A[i])
    and ray from p( v_s) in direction A[i] intersects S( v_g)
    and ray from p( v_s) in direction A[i].next intersects S( v_g)
    and SHOOTRAY( p( v_s) , angle_bisector( A[i], A[i].successor) , W) ∈ S( v_g) then
        U_p ← U_p ∪ ( A[i] + θ_max, A[i].next − θ_max)
    end if
end for
return U_p
```

Algorithm 3 LOCALPLANNER_POINTTOSEGMENT $(W, v_s, v_g, \theta_{\max})$

```
Require: v_s is a point node; v_g is a segment node.
U_p ← SAFEACTIONS_POINTTOSEGMENT ( W, v_s, v_g, θ_max)
if U_p = ∅ then
    return failure
else
    u ← arbitrary element of U_p
    return ( u)
end if
```

Algorithm 4 SAFEACTIONS_SEGMENTTOSEGMENT $(W, v_s, v_g, \theta_{\max})$

```
1:  U_s ← SAFEACTIONS_POINTTOSEGMENT ( W, src(v_s) , v_g, θ_max)
2:  U_t ← SAFEACTIONS_POINTTOSEGMENT ( W, tar(v_s) , v_g, θ_max)
3:  U ← empty set of actions
4:  for all intervals ( u_1, u_2) ∈ U_s ∩ U_t do
5:      if |u_2 − u_1| > 2θ_max then
6:          Q ← quadrilateral with vertices src(v_s), tar(v_s),
              SHOOTRAY( src(v_s) , d_2, W), and
              SHOOTRAY( tar(v_s) , d_1, W)
7:          for all vertices v ∈ W do
8:              if v ∈ Q then
9:                  goto 13
10:             end if
11:         end for
12:         U ← U ∪ ( u_1, u_2)
13:     end if
14: end for
15: return U
```

Algorithm 5 LOCALPLANNER_SEGMENTTOSEGMENT $(W, v_s, v_g, \theta_{\max})$

```
Require: v_s is a segment node; v_g is a segment node.
U ← SAFEACTIONS_SEGMENTTOSEGMENT ( W, v_s, v_g, θ_max)
if U = ∅ then
    return failure
else
    u ← arbitrary element of U
    return ( u)
end if
```

v_s to another segment node v_g, using a single action. This is a generalization of the problem in Section 4.4.1, in the sense that the starting position is now expanded from a single point into a interval of possible starting states. The intuition of our approach is to consider the intervals of safe actions for all of the (infinitely many) possible starting points along the continuum from src(v_s) to tar(v_s). If the intersection of *all* of these sets is non-empty, then the planner generates single action edge connecting v_s to v_g, labeled with an arbitrary action from the intersection.

As a starting point, note that it is trivially true that any action that is safe across the entire segment $S(v_s)$ must be safe from both src(v_s) and tar(v_s). To determine an initial set of candidate actions that satisfies this condition, we execute Algorithm 2 twice, and intersect the resulting intervals of safe actions. It remains to ensure that the result contains no actions that are unsafe from any interior point of $S(v_s)$.

To accomplish this, we consider the maximal intervals of actions safe from both src(v_s) and tar(v_s) in turn. For each such interval (u_1, u_2), we form a quadrilateral for which two vertices are src(v_s) and tar(v_s), and the remaining two vertices are the results of SHOOTRAY(src(v_s) , u_2, W) and SHOOTRAY(tar(v_s) , u_1, W). We then check the interior of this quadrilateral for the presence of obstacles. If any obstacle vertices are within the interior of this quadrilateral, then the entire interval can be discarded as unsafe. Figure 5 shows an example in which an interior obstacle eliminates an interval of actions that is safe from both src(v_s) and tar(v_s), but not from the interior of $S(v_s)$. Algorithm 4 shows the details, and Algorithm 5 shows the local planner itself.

The algorithm's two calls to Algorithm 2 each take $O(n \log n)$ time. To bound the number of intervals constructed by the loops on Algorithm 4, consider an environment in which every vertex $v \in W$, except the four endpoints of $S(v_s)$ and $S(v_g)$, lie between the two nodes. Given that at least three vertices are needed to form an obstacle that can split an interval and at most four intervals are formed from each obstacle, then there can be no more than $4(n-4)/3 \in O(n)$ intervals of actions generated between the two nodes. The test to determine whether a vertex lies inside the quadrilateral takes constant time,

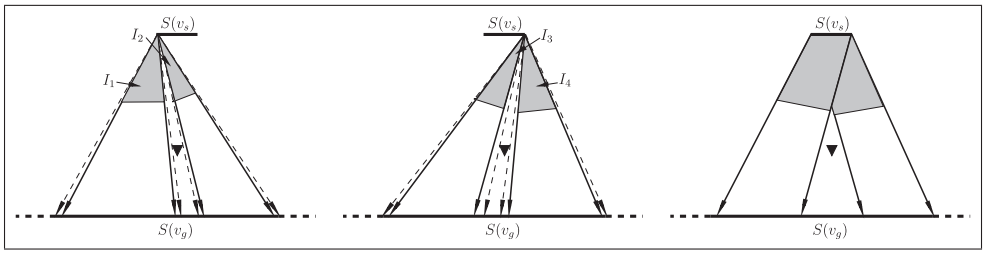

Fig. 5. An example segment-to-segment test. Top: There are two intervals I_1 and I_2 of safe actions from src(v_s). Middle: There are two such intervals I_3 and I_4 from tar(v_s). Bottom: The final result includes $I_1 \cap I_3$ and $I_2 \cap I_4$. A third non-empty interval, $I_1 \cap I_4$ is safe from both endpoints, but is correctly rejected by the algorithm because its quadrilateral contains an obstacle.

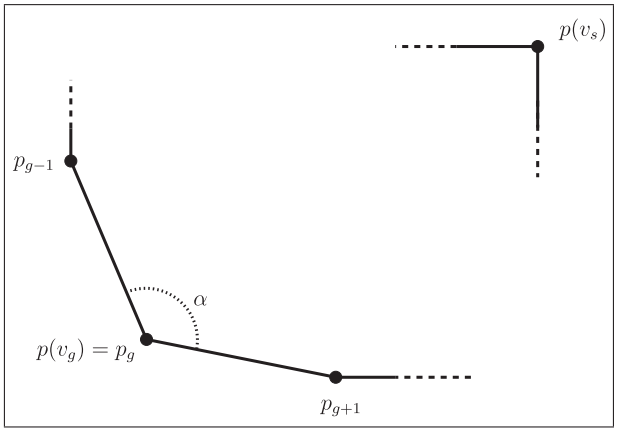

Fig. 6. The layout of points used in our corner-finding algorithm. A robot starting at $p(v_s)$ can navigate to $p(v_g)$, in spite of substantial motion errors, using our corner-finding algorithm.

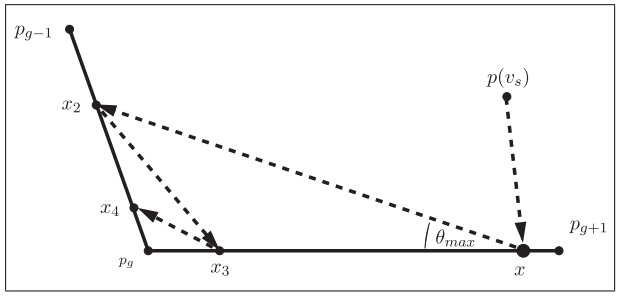

Fig. 7. Four actions generated by our corner-finding algorithm. The first action u_0 moves the robot safely to $p_g p_{g+1}$. Subsequent actions move the robot progressively closer to p_g.

therefore a maximum number of intervals in $O(n)$ leads to a total run time for Algorithm 5 in $O(n^2)$.

4.4.3. Corner-finding: point node to point node Suppose that our robot knows its own location, and would like to travel to a nearby convex vertex. How can the robot accomplish this, in spite of its motion uncertainty? Consider the environment portion depicted in Figure 6. The robot is known to be in a configuration at $p(v_s)$, and has a short-term goal of reaching $p(v_g)$. To compress the notation, in this section we write p_g to refer to $p(v_g)$. Let p_{g-1} and p_{g+1} denote the predecessor and successor vertices of p_g in a counterclockwise ordering of points, respectively. The segments $p_{g-1}p_g$ and $p_g p_{g+1}$ are the boundary edges between these vertices.

Corner-finding strategy The intuition of our approach is that the robot should first travel, in a single step, to either $p_{g-1}p_g$ or $p_g p_{g+1}$, and then make a series of back-and-forth transitions between those two segments, moving closer to p_g at each step. To simplify the description, we present the case in which the robot's first movement takes it from $p(v_s)$ to $p_g p_{g+1}$; the full algorithm considers both $p_g p_{g+1}$ and $p_{g-1}p_g$ as potential initial targets, making the obvious changes to the algorithms below.

After this first motion, the robot alternates between two actions:

1. whenever the robot is on $p_g p_{g+1}$, it chooses $u = \text{angle}(p_g - p_{g+1}) - \theta_{\max}$;
2. whenever the robot is on $p_{g-1}p_g$, it chooses $u = \text{angle}(p_g - p_{g-1}) + \theta_{\max}$.

Figure 7 illustrates a few steps of the robot's motions executing this strategy. These movements are carefully designed to reduce positional uncertainty by driving it toward p_g. The intent is for the robot move toward p_g as directly as possible, but to guarantee to make progress in spite of transition error, the robot instead aims $\theta_{\max}$ away from p_g. Note that we have described an infinite sequence of actions; we defer a discussion of when the robot should stop executing this sequence until after we describe the local planner.

Analysis of corner-finding The key question remaining with regards to this corner-finding strategy is to determine the conditions under which we can guarantee that the robot will arrive safely at p_g. The following lemma provides the characterization we need.

Lemma 1. *Let α denote the measure of angle formed at p_g with p_{g-1} and p_{g+1}. When executing the corner-finding algorithm described above, if (i) $0 < \alpha < \pi - 4\theta_{\max}$, (ii) the first transition, from $p(v_s)$ to x_1, is guaranteed to be collision-free, and (iii) the second transition, from x_1 to x_2, is guaranteed to be collision-free, then the robot is also*

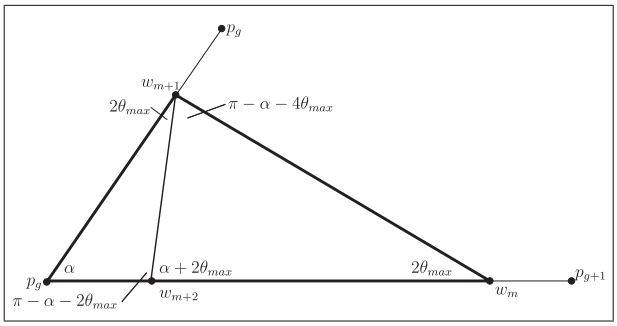

Fig. 8. Two triangles used in the proof of Lemma 1. We prove that $\triangle(w_{m+1}, w_{m+2}, p_g) \subset \triangle(w_m, w_{m+1}, p_g)$.

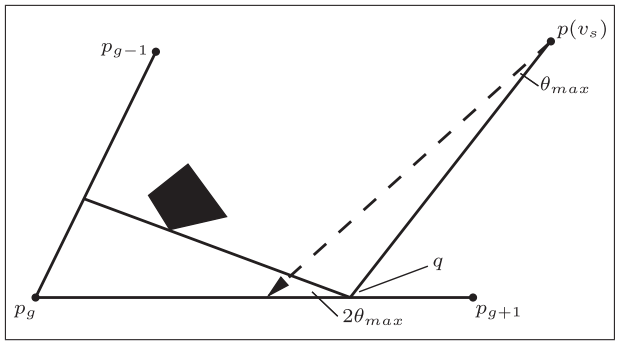

Fig. 9. Determining the outermost action u_0 onto segment $p_g p_{g+1}$ that guarantees that the second transition will reach $p_{g-1}p_g$. The dashed ray shows this action.

guaranteed to make the subsequent transitions to $x_3, \ldots, x_n$ without collision. Furthermore, the sequence $x_1, x_2, \ldots$ converges to p_g.

Proof. Use induction on the stage index k. As a base case, note that the conclusion is given for $k = 1$ and $k = 2$. For the induction step, assume that the statement is true for $k = m$ to show that it is true for $k = m + 1$. Refer to Figure 8. Let w_i denote the worst-case state stage i. That is, w_i is the outermost possible position for x_i. In the transition at stage m, the robot may pass through any point in the triangle formed by w_m, w_{m+1}, and p_g. By the inductive hypothesis, therefore, we know that the interior of this triangle does not contain any obstacles. Straightforward reasoning about the angles in this arrangement shows that $\angle(w_m, w_{m+1}, w_{m+2}) = \pi - \alpha - 4\theta_{\max}$, which by supposition is greater than zero. As a result, the triangle formed by w_m, w_{m+1}, and w_{m+2} is non-degenerate, and w_{m+2} is closer to p_g than w_m. This implies that the triangle formed by w_{m+1}, w_{m+2}, and p_g is fully contained within the triangle formed by w_m, w_{m+1}, and p_g. Since the latter triangle contains no obstacles, the former must also contain no obstacles. This ensures that the transition from x_{m+1} to x_{m+2} is collision free, completing the proof. □

The implication of Lemma 1 is that there are only three ways in which the corner-finding technique can fail: (1) if the angle at p_g is too big to ensure progress toward p_g; (2) if the robot collides with an obstacle on its initial translation, from $p(v_s)$ to $p_g p_{g+1}$; or (3) if the robot collides with an obstacle on its second transition, from $p_g pg + 1$ to $p_{g-1}p_g$. These conditions guide the design of our local planner, which we describe next.

Corner-finding local planner Note that if the robot is using this corner-finding approach, the only remaining choice for a local planner to make is to select the initial action u_0 that moves the robot from $p(v_s)$ to $p_g p_{g+1}$. The goal of our local planner, therefore, is to determine the set initial of actions for which conditions (2) and (3) above are met. The algorithm proceeds by finding sets of candidate values for u_0 that satisfy each of those two conditions, and then computing the intersection between those two sets.

If the intersection is non-empty, then the algorithm arbitrarily selects one of those actions, prepends it to the alternating corner-finding sequence, and returns success.

1. Condition (2): from $p(v_s)$ to $p_g p_{g+1}$. To find safe actions that reach $p_g p_{g+1}$ from $p(v_s)$, we note that this problem is equivalent to the problem addressed in Section 4.4.1 of finding a single action that reaches a segment node from a point node. Therefore, we execute Algorithm 2, with $p(v_s)$ as the starting point and all of $p_g p_{g+1}$ as the target.
2. Condition (3): from $p_g p_{g+1}$ to $p_{g-1}p_g$. The next step is find the set of possible initial actions u_0 for which the *second* transition is guaranteed to be collision free. Recall that after this first action, the robot will be somewhere along $p_g p_{g+1}$, and that the next action it selects will be nearly parallel to $p_g p_{g+1}$, but offset by $\theta_{\max}$. Therefore, it suffices for us to consider the set of triangles with one side parallel to the direction of this second action, and the other two sides contained in $p_g p_{g+1}$ and $p_{g-1}p_g$. The largest such triangle whose interior does not contain any obstacles determines the furthest from p_g that the robot can aim with action u_0, while ensuring that its second transition is collision-free. See Figure 9. Note that the extent of this largest triangle may be limited by obstacles (as seen in Figure 9) or by the lengths of $p_g p_{g+1}$ or $p_{g-1}p_g$.
   A method to compute this triangle appears as Algorithm 6. We first determine the direction of the second action (line 1 of Algorithm 6), and then shoot rays from each vertex in W in the opposite direction (Line 4 of Algorithm 6). To find the largest triangle, it is only necessary to note which of these rays lands closest to p_g along $p_g p_{g+1}$. This point becomes the outer endpoint of the interval of actions for which the second transition is safe (line 9 of Algorithm 6).
   Finally, we can state the local planner that considers these two sets of actions. See Algorithm 7. If the intersection of the two sets is non-empty, the algorithm generates an edge whose first action is chosen arbitrarily from the intersection, followed by the alternating

Algorithm 6 CORNERFINDING_SECONDSAFEACTIONS $(W, v_s, v_g, \theta_{\max})$

```
1: a ← angle(p_g − p_{g+1}) − 2θ_max
2: q ← p_{g+1}
3: for all vertices v ∈ W do
4:    z ← SHOOTRAY(v, a + π)
5:    if p ∈ p_g p_{g+1} and dist(p_g, q) ≤ dist(p_g, z) then
6:       q ← z
7:    end if
8: end for
9: return (angle(p(v_s) − p_g) + θ_max,
   angle(p(v_s) − q) − θ_max)
```

Algorithm 7 LOCALPLANNER_POINTTOPOINT $(W, v_s, v_g, \theta_{\max})$

```
Require: v_s is a point node; v_g is a point node.
1: if angle(p_{g−1}, p_g, p_{g+1}) ≥ π − 4θ_max then
2:    return failure
3: else
4:    U_1 ← SAFEACTIONS_POINTTOSEGMENT
      (W, v_s, v_g, θ_max)
5:    U_2 ← CORNERFINDING_SECONDSAFEACTIONS
      (W, v_s, v_g, θ_max)
6:    if U_1 ∩ U_2 = ∅ then
7:       return failure
8:    else
9:       u_0 ← arbitrary element of U_1 ∩ U_2
10:      u_1 = angle(p_g − p_{g+1}) − θ_max
11:      u_2 = angle(p_g − p_{g−1}) + θ_max
12:      return (u_0, u_1, u_2, u_1, u_2, ...)
13:   end if
14: end if
```

sequence of back-and-forth actions described above. If the intersection is empty, we return failure.

Corner-finding with finite action sequences The final detail to consider with regards to this corner-finding approach is the fact that the action sequence generated at line 12 of Algorithm 7 is an infinite sequence. The states reached by this sequence converge to p_g in the limit, but cannot guarantee to reach p_g exactly. The algorithm itself stores a finite representation of this repeating sequence, but in practice the robot must truncate it at some point, either to continue on to the next graph edge in the overall plan, or to terminate near x_G. There are at least two reasonable choices for when to terminate the corner-finding oscillations:

1. We can choose a fixed number N of iterations. The algorithm converges sufficiently quickly that a relatively small N is sufficient to move the robot closer to p_g than typical machine precision is able to represent.
2. After constructing the complete navigation plan, we can determine when to terminate corner-finding based on its context within that plan. In the general case, when the corner-finding edge is *not* the final edge in the plan, then we can stop whenever w_k is sufficiently close to p_g that the remaining distance is irrelevant to the success of the plan. If the corner-finding edge is the final edge in the plan (so that $p_g = x_G$), then the robot can terminate as soon as $\text{dist}(p_g, w_k) < \delta$.

4.4.4. Corner-finding: segment node to point We also use a local planner that builds edges from segment nodes to point nodes, using a variation on the corner-finding technique we introduced in Section 4.4.3. The basic motion strategy—to plan a single action to reach an environment edge incident to the target vertex, and then to alternate between the two incident edges—remains the same. As in Section 4.4.3, we assume for simplicity that the robot's first motion goes to $p_g p_{g+1}$; the full algorithm considers both $p_g p_{g+1}$ and $p_{g-1} p_g$ as initial targets.

Note that Lemma 1 still holds in this context, and that Algorithm 6 can still determine the initial actions for which the robot's second transition is safe. Therefore, the key extension we need is a way to determine the set of initial actions for which the robot's *first* transition from $S(v_s)$ is guaranteed to reach $p_g p_{g+1}$. Conveniently, Algorithm 4, which we have already described in Section 4.4.2, computes precisely this set of actions.

The resulting local planner, which closely parallels the local planner from Section 4.4.3, appears as Algorithm 8. Its $O(n^2)$ run time is dominated by the call to Algorithm 4.

4.4.5. Long hallways: point node to point node with oscillation and corner-finding Our fifth and final local planner is an alternative way to generate edges between pairs of point nodes. The motivation for this alternative is a weakness in our approach to segment node generation, specifically in environments containing long edges with no environment vertices in between. An example of this problematic structure is a long "hallway", in which there are four vertices at the ends and none in the middle, as seen at the far right side of Figure 1. In this scenario, Algorithm 1 generates segment nodes nearly as long as the hallway itself. Using the local planners described above, it is extremely difficult to generate outgoing edges from such nodes. As a consequence, the global planner often fails to find plans that traverse such hallways. To overcome this shortcoming, we provide an algorithm whereby the robot attempts to make progress by alternating motions between the hallway walls.

The intuition is to use the *forward projection* of an action from a starting segment, which is simply the far boundary of the corresponding error cone. The algorithm considers two possible initial actions, offset from $\text{angle}(p(v_g) - p(v_s))$ by $\pi/4$ in either direction. For each of these initial actions, the algorithm proceeds by computing a series of forward projections under actions chosen to be $\pi/4$ radians away from the edge reached by the previous forward projection. This process continues until either (1) one of the

Algorithm 8 LocalPlanner_SegmentToPoint($W, v_s, v_g, \theta_{\max}$)

Require: v_s is a segment node; v_g is a point node.

if angle$(p_{g-1}, p_g, p_{g+1}) \geq \pi - 4\theta_{\max}$ **then**
  return failure
else
  $U_1 \leftarrow$ SafeActions_SegmentToSegment $(W, v_s, v_g, \theta_{\max})$
  $U_2 \leftarrow$ CornerFinding_SecondSafeActions $(W, \text{src}(v_s), v_g, \theta_{\max})$
  $U_2 \leftarrow U_2 \cap$ CornerFinding_SecondSafeActions $(W, \text{tar}(v_s), v_g, \theta_{\max})$
  if $U_1 \cap U_2 = \varnothing$ **then**
    return failure
  else
    $u_0 \leftarrow$ arbitrary element of $U_1 \cap U_2$
    $u_1 = \text{angle}(p_g - p_{g+1}) - \theta_{\max}$
    $u_2 = \text{angle}(p_g - p_{g-1}) + \theta_{\max}$
    return $(u_0, u_1, u_2, u_1, u_2, \dots)$
  end if
end if

forward projections is unsafe, in the sense of containing an environment vertex or reaching two or more different environment edges or (2) the algorithm reaches a predetermined limit of l iterations. We use this limit to ensure that the algorithm will halt within a reasonable time. After this process is complete, we invoke Algorithm 8 (LocalPlanner˙SegmentToPoint) from the last safe forward projection, attempting to attempt to reach $p(v_g)$. If this corner-finding step succeeds, we combine the two partial plans to form a complete edge from v_s to v_g. For brevity, we omit pseudocode for this local planner. This algorithm's run time is $O(n^2)$, dominated by the call to Algorithm 8.

4.5. Prioritizing the search

Sections 4.3 and 4.4 described a directed graph with as many as $O(n^4)$ nodes. Attempting to generate edges between each pair of those nodes requires the execution of several local planners. As a result, it is computationally expensive to perform a brute force approach that executes all of the local planners for all node pairs. Instead, we order the pairs of nodes before attempting to generate any edges. The goal is to choose an order of the node pairs that favors pairs that (1) are likely to admit an edge between them and (2) are likely to be useful for navigation between the given starting node v_S and goal node v_G.

Specifically, we use a *priority function* that accepts two node pairs $\left(v_s^{(1)}, v_g^{(1)}\right)$ and $\left(v_s^{(2)}, v_g^{(2)}\right)$ as its input, and returns a Boolean value indicating whether the first pair should be considered before the second pair. The priority function we propose applies a sequence of tests, continuing to each subsequent test only when the both node pairs have equal values for the previous test: (1) prefer the pair for which the final node $v_g^{(i)}$ is a point node; (2) prefer the pair for which the final node $v_g^{(i)}$ is closer in geodesic distance—that is, shortest path distance within W—to the global goal v_G; (3) prefer the pair for which the final node $v_g^{(i)}$ is smaller; (4) prefer the pair for which the starting node $v_s^{(i)}$ is closer, in geodesic distance, to the global goal v_G; (5) prefer the pair for which the starting node $v_s^{(i)}$ is smaller.

The intuition of these tests is to drive the search toward nodes that are nearer to the goal, noting that it is more challenging to generate outgoing edges from larger nodes. The key idea is that evaluating this priority function requires less computational expense than executing the local planners from Section 4.4.

To enable this ordering to be enforced efficiently, we preprocess the environment by constructing its visibility graph and the all-pairs shortest-path matrix for this visibility graph. This data structure enables constant-time lookups of the shortest path distance between any pair of environment vertices. In the case of point nodes, geodesic distances are computed directly. In the case of segment nodes, we overestimate the distance by using the endpoints of the environment edge on which the node lies. Specifically, for distances involving a segment node v, we use the mean of two distances: (1) the distance from the nearest endpoint of the edge containing $S(v)$ to src(v); and (2) the distance from the nearest endpoint of the edge containing $S(v)$ to tar(v). Note that, in the case of a small segment node near one of its edge's endpoints, the vertices used for both (1) and (2) may be the same.

In addition to this ordering, we also implement a filtering technique in which we discard outright any node pairs that are separated by three or more turns (determined by a count of intermediate vertices) in the shortest path between them within W. Such node pairs are unlikely to be connected by any of our local planners. Instead, we rely on the global planner to generate multi-edge plans to traverse those kinds of environment regions.

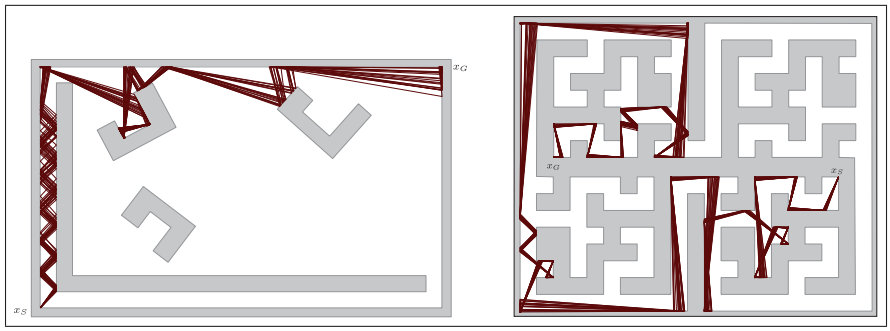

Fig. 10. Two plans generated by our algorithm. For each, twenty simulations of the plan are shown in parallel. Left: A non-rectilinear environment, in which we used $\theta_{\max} = \pi/36$. Right: A large-scale maze, in which we used $\theta_{\max} = \pi/72$.

4.6. Global planner

Finally, we can assemble the components introduced in the previous sections into our complete navigation planner.

The algorithm is a specialized forward graph search that maintains three primary data structures. (1) A *navigation graph* G, with the nodes introduced in Section 4.3, but initially having no edges. (2) A *unconnected set of nodes* Z. Each node v for which there does not yet exist a directed path $v_S \longrightarrow \cdots \longrightarrow v$ through G is included in Z. The algorithm starts with $Z = V(G) - \{v_S\}$. (3) A *priority queue* Q *of pairs of nodes* in G, ordered by the priority function introduced in Section 4.5. These represent node pairs for which local planners will be attempted in the future. This queue is seeded with all pairs whose starting node is v_S.

The algorithm proceeds by repeatedly extracting node pairs (v_s, v_g) from Q and executing each of the applicable local planners described in Section 4.4 to attempt to connect v_s to v_g. If any of local planners succeed, we add the corresponding edge to G, remove v_g from Z, and insert the all of node the pairs $\{v_g\} \times Z$ into Q.

The algorithm continues until one of two termination conditions occurs. If v_G is removed from Z, then a complete plan has been found, and the action sequence can be extracted from the sequence of edges connecting v_S to v_G. If x_G is never removed from Z, the algorithm terminates with a failure when Q becomes empty.

Extension 2 illustrates the execution of this algorithm. In that video, the yellow marks and light grey arrows show nodes that have been connected to v_S (that is, those that are *not* in Z) and the blue marks and green marks show the starting and goal nodes, respectively, that the algorithm is attempting to connect at each iteration.

5. Implementation and experiments

We have implemented this algorithm both in simulation and on a physical robot platform. The implementation is in C++ and uses the Computational Geometry Algorithms Library (CGAL) for exact and robust geometric computations. This section presents the results of our evaluation of the algorithm using this implementation.

5.1. Simulations

To illustrate the algorithm's behavior, we executed it on several representative planning problems. For each one, we used our algorithm to find a navigation plan and simulated the execution of that plan using a pseudo-random number to generate each θ_k.

1. Figure 1 is a relatively simple rectilinear environment with 44 vertices. We refer to this environment again in Sections 5.1.1 and 5.1.2 to evaluate the algorithm quantitatively.
2. Figure 10 (left) is a non-rectilinear environment with 42 vertices, divided by an obstacle inducing two very long halls. In this experiment, we used $\theta_{\max} = \pi/36 \approx 5^\circ$ for the maximum error at each step. The effect of this uncertainty is evident in the cones created by the robot's different paths. Our planner required 364 edges and 114,847 edge connection attempts to generate the solution shown. Extension 3 shows a simulation of this plan.
3. Figure 10 (right) is an environment containing 172 vertices and several instances of long halls and T-junctions. This solution used $\theta_{\max} = \pi/72 \approx 2.5^\circ$ and required 193 edges and 24,004 connection attempts. Extension 4 shows a simulation of this plan. This example illustrates the ability of our algorithm to generate paths that span long distances, far beyond what this robot could achieve using a naïve navigation approach.

In each of these simulations, the robot successfully reached its goal on every attempt, corroborating the correctness of the algorithm.

5.1.1. Quantitative analysis of success rates To measure the algorithm's effectiveness quantitatively, we executed it in the environment depicted in Figure 1 for values of $\theta_{\max}$ between 0.01 and 0.1 in increments of 0.01. For each $\theta_{\max}$, we attempted to generate a navigation plan from each convex vertex to each other convex vertex, for total of 576 problem instances. Figure 11 plots the success rate of the algorithm under these tests.

For $\theta_{\max} = 0.01$ and $\theta_{\max} = 0.02$, the planner successfully found plans connecting all pairs of point nodes. The

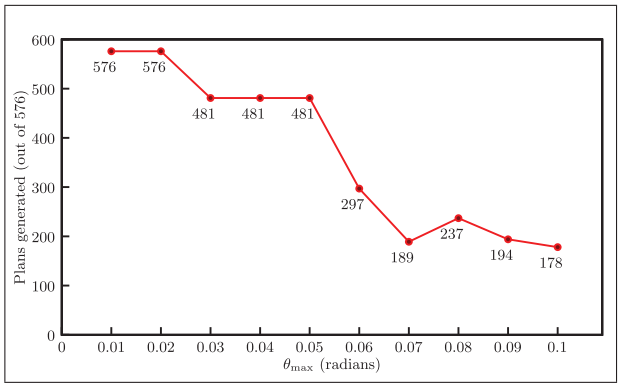

Fig. 11. Success rates for our algorithm in the environment depicted in Figure 1, across all 576 pairs of convex vertices.

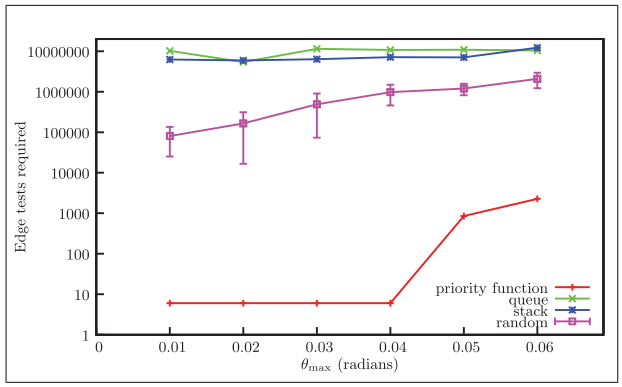

Fig. 12. Comparison of algorithm efficiency, measured by edge tests, for four different node pair ordering schemes, for the problem depicted in Figure 1. For the randomized algorithm, we plot the mean and show the standard deviation using error bars. Note the logarithmic scale on the vertical axis.

remaining results largely confirm the intuition that, as θ_{max} decreases, the algorithm is more likely to succeed. However, the success rate is not strictly a monotonically non-increasing function of θ_{max}. This apparent discrepancy can be explained by the fact that the algorithm uses θ_{max} not just for connecting segment nodes, but also for generating those nodes. If, for a certain combination of W and θ_{max}, the delimiting points happen to fall in unhelpful locations, then the algorithm may fail even in cases for which it succeeds for larger values of θ_{max}. In the data shown in Figure 11, this phenomenon appears to occur for $\theta_{max} = 0.07$. We briefly discuss the possibilities for choosing better sets of segment nodes in Section 6.

5.1.2. Evaluation of the priority function To evaluate the effectiveness of our priority function, we solved the planning problem depicted in Figure 1 using both our algorithm and similar planners that use a stack, a first-in first-out queue, and a random ordering of the node pairs, for six distinct values of θ_{max}. Figure 12 plots the number of edge tests, each of which consists of running all of the relevant local planners, used by each algorithm. The other parts of the algorithm have negligible run time compared with these tests. For the randomized algorithm, we performed 10 trials for each θ_{max} and show the standard deviation using error bars. The results, which are plotted on a logarithmic scale, show that our prioritized ordering leads to orders of magnitude of improvement over straightforward approaches.

5.2. Physical implementation

To confirm the plausibility of using this algorithm on a real robot, we implemented it on an iRobot Create differential drive platform. These robots have two forward-facing bump sensors and a side-oriented single-point infrared range sensor with a maximum range of a few centimeters, along with a handful of other sensors that we ignored for this experiment. We did not add any external sensors to the robot.

To implement the model introduced in Section 3 on this robot, we exploit the fact that each action in the plan is a safe action. As a result, as the robot executes the plan, it always knows which environment edge it is touching, even when it does not know its position along that edge. Based on this information, we use the robot's infrared sensor to implement a "rotate-to-parallel" subroutine that attempts to align the robot's facing with the adjacent environment boundary. From this known orientation, the robot rotates the remaining angle required to reach the direction required for the next action in the plan. Note that both of these rotation steps are subject to errors. We account for both error sources by setting $\theta_{max} = 10^{\circ}$, a bound which comfortably exceeds the errors we observed in most of our tests.

Figure 13 shows the synthetic environment used in our tests. We generated two plans to navigate within this environment and executed each of them with the robot 10 times. Extension 5 shows a sample execution. Details about the setup and results appear in Table 1. These results show that, in 19 of the 20 trials, the robot successfully reached its goal. The lone failure occurred as a result of an under-rotation of approximately 30° in one of the rotate-to-parallel operations, presumably due to unusually large sensor noise. In this case, the robot arrived at the location marked "C" in Figure 13. In each case in which the assumptions of the algorithm were satisfied, the robot successfully reached its goal.

6. Discussion and conclusion

We have presented a planning algorithm which generates navigation plans that can be executed by very simple robots, including those having only a compass, map, and contact sensor. The planner considers the bound on error in the robot's actions and chooses plans to move the robot reliably through its environment in spite of these errors. However, a number of interesting questions remain unanswered.

Table 1. Results of our physical experiments.

x_S	x_G	$\theta_{\max}$	Computed plan	Number of actions	Successes	Failures	Travel time mean (SD)
A	B	10°	A → D → B	9	9	1	90 s (1.06)
B	A	10°	B → E → D → A	10	10	0	105 s (1.41)

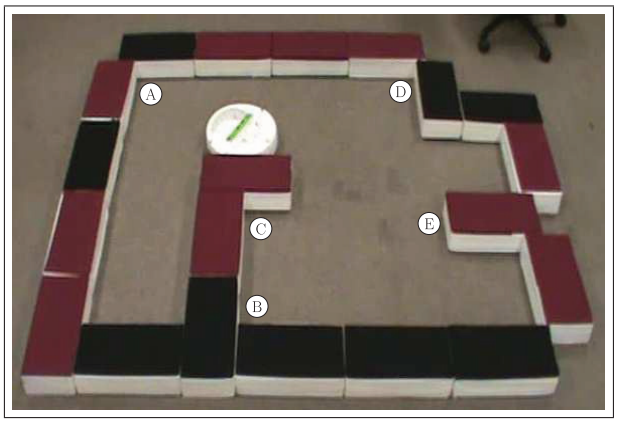

Fig. 13. The artificial environment used in our experiments.

First, observe that our algorithm relies on an assumption that any errors in the robot's heading are bounded by a constant. However, in many robot systems, such as a typical differential drive robot with no compass, the heading error increases as a function of the distance traveled, resulting curved paths. For systems in which this systematic drift is not negligible, somewhat more complex geometric algorithms—using error cones with appropriately curved boundaries—would be necessary.

Second, note that the algorithm presented in this paper is sound, in the sense that any plans it generates are guaranteed to get the robot to its goal, provided that the motion error remains within the $\theta_{\max}$ bound. However, it is possible that the algorithm will fail on problem instances for which a solution does exist. Hence, our algorithm is not complete. Finding an efficient, complete algorithm for this problem appears to be a very challenging open problem. We have identified three primary issues that would need to be addressed to move toward a complete algorithm for this problem.

6.1. Selection of segment nodes

Our algorithm uses a specific, finite set of segment nodes. This set has proven practically useful, but it remains possible that an algorithm using a finer division of the environment boundary would succeed for some problem instances in which our algorithm fails. One possible approach would be to supplement or replace the delimiting points proposed in Section 4.3 with an incremental sequence of samples along each edge. In this scenario, whenever Q becomes empty, the algorithm would generate additional segment nodes by inserting new delimiting points. This could lead to a notion of resolution completeness, in the sense that, if the sequence of samples is dense, this approach will eventually generate a segment node arbitrarily close to any possible segment node that a solution might require.

6.2. Selection of local planners

Our algorithm also uses a specific, finite set of local planners. Certainly the addition of more local planners can only improve the algorithm's ability to generate successful plans. Note, however, that a tradeoff exists between the richness of the set of segment nodes and the need for powerful local planners. At one extreme, if we use a sufficiently rich set of segment nodes, then the single-step local planner proposed in Section 4.4.2 would be sufficient to produce a complete global navigation plan, without any other local planners. At the other extreme, if we had access to an "ideal" local planner that always generates an edge if one exists, then the algorithm could simply use this planner to connect v_S to v_G, without concerning itself with any intermediate nodes. It seems likely that the most efficient complete algorithm would exist somewhere between these two extremes.

6.3. Unsafe actions

Finally, and perhaps most importantly, every action generated by our algorithm is a safe action. Recall that safe actions are those whose forward projections reach only a single edge of the environment boundary. Figure 14 shows an example problem instance in which there are no safe actions from x_S. For this problem, any algorithm similar to ours—regardless of any improvements to GENERATENODES, and regardless of the addition of any new local planners—will fail. However, Figure 14 also shows a simple repeating sequence of four actions that does solve this problem. This example illustrates the need for any complete algorithm to keep track of forward projections that span multiple environment edges. In the context of our algorithm, this corresponds to a search through a much larger graph, in which each node represents a collection of multiple possible nodes in the current graph G. It is not clear how to plan efficiently over this exponentially larger graph.

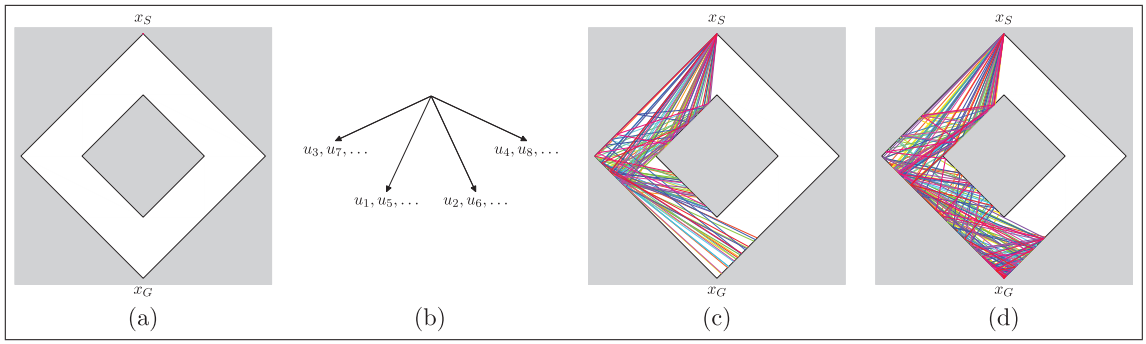

Fig. 14. An illustration of the limitations of our approach. (a) An environment for which, when $\theta_{\max} = 0.35 \approx 20^\circ$, there are no safe actions from x_S. (b) A repeating sequence of four (unsafe) actions which, when executed in sequence, solve the navigation problem by bringing the robot arbitrarily close to x_G. (c) After one complete cycle through this sequence, the robot has made progress toward the goal, but cannot be certain of its position. (d) After 4 complete cycles (16 actions), the robot's position is within a small radius of the goal.

Funding

This work is partially supported by a grant from the University of South Carolina, Office of Research and Health Sciences Research Funding Program, from the US National Science Foundation (grant number IIS-0953503) and from the Defense Advanced Research Projects Agency (grant number N10AP20015).

Note

1. This paper focuses on point robots only for simplicity of exposition; the results can be extended to disc-shaped robots with non-zero radius in a straightforward way, by using as W a polygonal approximation of the free portion of the robot's configuration space.

References

Acar EU and Choset H (2001) Complete sensor-based coverage with extended-range detectors: A hierarchical decomposition in terms of critical points and Voronoi diagrams. In: *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems*.

Agha-mohammadi A, Chakravorty S and Amato NM (2012) On the probabilistic completeness of the sampling-based motion planning methods under uncertainty. In: *Proceedings IEEE International Conference on Robotics and Automation*.

Akella S, Huang W, Lynch KM and Mason MT (2000) Parts feeding on a conveyor with a one joint robot. *Algorithmica* 26(3): 313–344.

Akella S and Mason M (1998) Posing polygonal objects in the plane by pushing. *The International Journal of Robotics Research* 17(1): 70–88.

Berretty RP, Goldberg K, Overmars M and van der Stappen F (2001) Trap design for vibratory bowl feeders. *The International Journal of Robotics Research* 20(11): 891–908.

Blum A, Raghavan P and Schieber B (1997) Navigating in unfamiliar geometric terrain. *SIAM Journal on Computing* 26(1): 110–137.

Blum M and Kozen D (1978) On the power of the compass (or, why mazes are easier to search than graphs). In: *Proceedings IEEE Symposium on Foundations of Computer Science*. pp. 132–142.

Choset H and Burdick J (2000a) Sensor based motion planning: Incremental construction of the hierarchical generalized Voronoi graph. *The International Journal of Robotics Research* 19(2): 126–148.

Choset H and Burdick J (2000b) Sensor based motion planning: The hierarchical generalized Voronoi graph. *The International Journal of Robotics Research* 19(2): 96–125.

Deng X, Kameda T and Papadimitriou CH (1998) How to learn an unknown environment I: The rectilinear case. *Journal of the ACM* 45(2): 215–245.

Donald BR (1995) On information invariants in robotics. *Artificial Intelligence* 72: 217–304.

Erdmann M and Mason MT (1988) An exploration of sensorless manipulation. *IEEE Transactions on Robotics and Automation* 4(4): 369–379.

Erdmann MA (1986) Using backprojections for fine motion planning with uncertainty. *The International Journal of Robotics Research* 5(1): 19–45.

Erdmann MA (1995) Understanding action and sensing by designing action-based sensors. *The International Journal of Robotics Research* 14(5): 483–509.

Erickson L, Knuth J, O'Kane JM and LaValle SM (2008) Probabilistic localization with a blind robot. In: *Proceedings IEEE International Conference on Robotics and Automation*.

Goldberg KY (1993) Orienting polygonal parts without sensors. *Algorithmica* 10: 201–225.

Hauser K (2010) Randomized belief-space replanning in partially-observable continuous spaces. In: *Proceedings of the Workshop on the Algorithmic Foundations of Robotics*.

Kamon I and Rivlin E (1997) Sensory-based motion planning with global proofs. *IEEE Transactions on Robotics and Automation* 13(6): 814–822.

Kamon I, Rivlin E and Rimon E (1999) Range-sensor based navigation in three dimensions. In: *Proceedings IEEE International Conference on Robotics and Automation*.

Katsev M, Yershova A, Tovar B, Ghrist R and LaValle SM (2011) Mapping and pursuit-evasion strategies for a simple wall-following robot. *IEEE Transactions on Robotics* 27(1): 113–128.

Kloetzer M and Belta C (2007) Temporal logic planning and control of robotic swarms by hierarchical abstractions. *IEEE Transactions on Robotics* 23(2): 320–330. DOI: 10.1109/TRO.2006.889492.

Lazanas A and Latombe JC (1992) Landmark-based robot navigation. In: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.

Lewis JS and O'Kane JM (2010) Guaranteed navigation with an unreliable blind robot. In: *Proceedings IEEE International Conference on Robotics and Automation*.

Lewis JS and O'Kane JM (2012) Reliable indoor navigation with an unreliable robot: Allowing temporary uncertainty for maximum mobility. In: *Proceedings IEEE International Conference on Robotics and Automation*.

Lozano-Pérez T, Mason MT and Taylor RH (1984) Automatic synthesis of fine-motion strategies for robots. *The International Journal of Robotics Research* 3(1): 3–24.

Lumelsky VJ and Tiwari S (1994) An algorithm for maze searching with azimuth input. In: *Proceedings IEEE International Conference on Robotics and Automation*, pp. 111–116.

Mason M (1993) Kicking the sensing habit. *AI Magazine* 14(1): 58–59.

McLurkin J, Prabhu S and Li W (2012) Hexagonal lattice formation in multi-robot systems. In: *Proceedings of the International Symposium on Distributed Autonomous Robotic Systems*.

Mier-y-Teran-Romero L, Forgoston E and Schwartz IB (2012) Coherent pattern prediction in swarms of delay-coupled agents. *IEEE Transactions on Robotics* 28(5): 1034–1044. DOI: 10.1109/TRO.2012.2198511.

Moll M and Erdmann M (2002) Manipulation of pose distributions. *The International Journal of Robotics Research* 21(3): 277–292.

Ó Dúnlaing C and Yap CK (1982) A retraction method for planning the motion of a disc. *Journal of Algorithms* 6: 104–111.

O'Kane JM and LaValle SM (2007) Localization with limited sensing. *IEEE Transactions on Robotics* 23: 704–716.

O'Kane JM and LaValle SM (2008) On comparing the power of robots. *The International Journal of Robotics Research* 27(1): 5–23.

Prentice S and Roy N (2009) The belief roadmap: Efficient planning in belief space by factoring the covariance. *The International Journal of Robotics Research* 28(11): 1448–1465. DOI: 10.1177/0278364909341659.

Roy N and Thrun S (1999) Coastal navigation with mobile robots. In: *Advances in Neural Processing Systems*, pp. 1043–1049.

Szirmay-Kalos L and Marton G (1998) Worst-case versus average case complexity of ray-shooting. *Computing* 61(2): 103–131.

Tovar B, Guilamo L and LaValle SM (2004) Gap Navigation Trees: minimal representation for visibility-based tasks. In: *Proceedings of the Workshop on the Algorithmic Foundations of Robotics*.

van der Stappen AF, Berretty RP, Goldberg K and Overmars MH (2000) Geometry and part feeding. In: *Sensor Based Intelligent Robots*, pp. 259–281.

Whitney DE (1986) Real robots don't need jigs. In: *Proceedings IEEE International Conference on Robotics and Automation*.

Appendix: Index to Multimedia Extensions

The multimedia extension page is found at http://www.ijrr.org

Table of Multimedia Extensions

Extension	Type	Description
1	Video	Simulated execution of the plan depicted in Figure 1.
2	Video	Visualization of the planning process.
3	Video	Simulated execution of the plan depicted in Figure 10 (left).
4	Video	Simulated execution of the plan depicted in Figure 10 (right).
5	Video	Execution on a differential drive robot.