Reliable Indoor Navigation with an Unreliable Robot: Allowing Temporary Uncertainty for Maximum Mobility

Jeremy S. Lewis and Jason M. O'Kane

***Abstract*— In this work we consider a navigation problem for a very simple robot equipped with only a map, compass, and contact sensor. Our prior work on this problem uses a graph to navigate between the convex vertices of an environment. In this paper, we extend this graph with the addition of a new node type and four new edge types. The new node type allows for more uncertainty in robot position. The presence of one of these new edge types guarantees reliable transitions between these nodes. This enhanced graph enables the algorithm to navigate environment features not solvable by our previous algorithm, including T-junctions and long halls. We also present a heuristic to accelerate the planning process by prioritizing the promising edge tests to perform. Our heuristic effectively focuses the search and qualitative data show that it computes plans with much less computational effort than a naïve approach. We describe a simulated implementation of the algorithm that finds paths not previously possible, and a physical implementation that demonstrates the feasibility of executing those plans in practice.**

I. Introduction

The ability to navigate in cluttered environments is fundamental to many robotic tasks. This is a challenging goal for a mobile robot due to the combined difficulties of both finding a plan to move a robot from a beginning location to a goal location and executing that plan in consideration of uncertainty in both sensing and actuation. While it is typical in navigation research to solve these problems separately, this approach does not allow for plans with actions specifically intended to reduce uncertainty. In this paper we consider a navigation problem for a very simple robot, having only a map, compass, and contact sensor. There exists uncertainty in both sensing and actuation. Our planner directly considers the robot's uncertainty to generate plans containing steps that move the robot toward the goal, reduce uncertainty when necessary, or both.

In our previous work [8] on this problem, we showed that in certain conditions, movements by this robot can decrease uncertainty; however, that algorithm tends to fail at predictable environment features such as T-junctions and long halls. Our key insight is to allow the robot's state uncertainty to temporarily grow from a single state to a set of possible states. This allows the robot to traverse those features that stymied our original algorithm. To achieve this, we search for plans on a graph in which each node represents a set of states, in contrast to our prior work that considered only singleton sets.

J. S. Lewis and J. M. O'Kane are with the Department of Computer Science and Engineering, University of South Carolina, 301 Main St., Columbia, SC 29208, USA.{lewisjs4, jokane}@cse.sc.edu

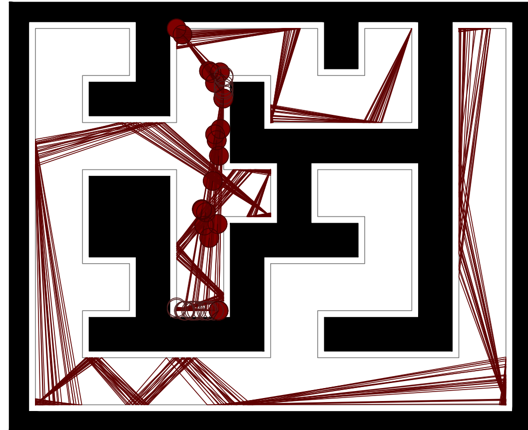

Fig. 1. A plan generated by our algorithm. The figure shows 20 simultaneous simulations of a robot executing that plans, each receiving different sensor noise. Our algorithm guarantees that the robot will reach its goal in spite of these errors.

Specifically, in our new graph, nodes represent continua of points along the boundary of the environment. Because there are infinitely many potential nodes in such a graph, we choose a finite subset of those potential nodes, based on a visible-vertex analysis of the environment that accounts for the robot's motion error. This method generates nodes useful for navigating around the reflex vertices that constitute the problematic features we aim to overcome. This method creates $O(n^4)$ nodes in an environment with n vertices.

Each edge between these nodes is labeled with some sequence of actions the robot can execute to ensure safe movement from the set of possible states represented by one node to the set of possible states represented by another. The sets of states are represented as line segments and singleton sets as points. We present the five algorithms to generate such labeled edges: (1) a point to point test with corner-finding, (2) a direct point to segment test, (3) a direct segment to segment test, (4) a segment to point test with corner-finding, and (5) a point to point test for long hallways.

The planning algorithm is a process of constructing these edges. However, it is impractical to exhaustively test each of the $O(n^8)$ potential connections between the nodes. Instead, we use a heuristic to order the node pairs considered for connection by their applicability to the overall plan and by their potential for success in one of our edge tests. The heuristic considers the progress made by each potential edge toward the goal (measured in geodesic distance), while preferring transitions between nodes with relatively little

uncertainty. Our heuristic also eliminates altogether nodes which have no chance for success in our edge tests. We present quantitative data illustrating the superior performance of our planner's heuristic-driven search in comparison to randomized, stack-ordered, and queue-ordered searches.

The remainder of the paper is structured as follows. In Section II, we discuss related research. Section IV gives a formal problem statement. Our algorithm appears in Section V, and we present an implementation, in simulation, in Section VI. Section VII concludes the paper with discussion and a preview of future work.

II. Related work

The goal of simplifying sensing and actuation in robotic systems, retaining the capability to solve meaningful problems, is not new. Many tasks are considered with this approach including: manipulation [9] navigation [7], and mapping [14]. Additionally, the question of task completion under a minimal sensor model has been considered [1], [3], [5], [11]. Often the idea behind such an approach is that by reducing the power of the robot, a greater understanding of the problem is obtained.

Our planning algorithm uses ideas introduced by Erickson, Knuth, O'Kane, and LaValle [6]. That research also uses the idea of an error cone to model uncertainty in rotation, which moves a robot's state probability distribution from one distribution to another. In solving a global active localization problem, the authors use a system of actions to drive a robot's state probability toward a single cell of a coarse discretization of the environment. The differences in our work are that we are solving a navigation problem, we use nondeterministic reasoning to achieve guarantees of success, and we use precise geometry rather than a discretization of the environment. Our high-level transitions are also very much in the spirit of pre-image backchaining introduced by Lozano-Pérez, Mason, and Taylor [9]. Our corner-finding algorithm is very similar to the notion of a fine-motion strategy countering positional uncertainty. The lessons of these problems allow us to guarantee, if our algorithm generates a plan and the robot's sensor noise remains within the bounds of the model, that the robot will successfully navigate from start to goal.

The approach we use in our planner parallels the idea of landmark-based navigation introduced by Lazanas and Latombe [7]. Their work assumes the robot's sensor model is error-free while in the presence of landmarks, whereas landmarks give our robot no additional sensing capabilities. Their robot model also assumes a goal detector which tells it exactly when it reaches its goal. We do not have sensors to detect goals nor any landmarks; instead, we exploit the geometric features of the environment to reduce uncertainty without sensing.

Our work considers the uncertainty of the robot as a guide in path planning. This approach is similar to planning using coastal navigation techniques [12]—that is moving from one landmark to another using range sensors to detect landmarks. Our model is quite different however in that we

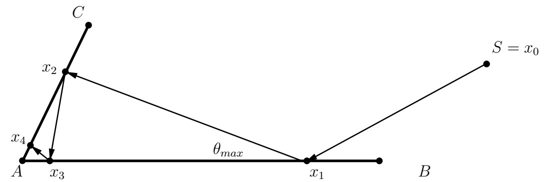

Fig. 2. Corner-finding allows the robot to travel from S to A, in spite of rotation errors up to the error bound θ_{max}.

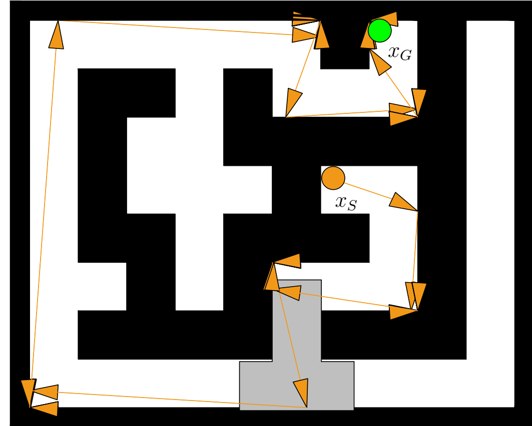

Fig. 3. A plan to travel from state x_S to state x_G, generated by the authors' previously published algorithm. However, that algorithm fails to find a path to return to x_S from x_G due to a T-junction highlighted by shaded area.

consider having only a map, a noisy compass, and a contact sensor. There has also been prior work considering robot models similar to ours for other tasks [10], [11]. However, we consider a more realistic model for robot motion that includes substantial errors and show that solutions can still be generated for many navigation problems. The motivation behind this is the desire to solve problems such as this with simple robots.

III. Summary of authors' prior work

The results in this paper extend the authors' prior work [8] on the same problem. The intuition of that approach is that, in the right circumstances, a mobile robot can reduce its positional uncertainty via a sequence of back-and-forth motions that drive it toward a corner in the environment. Figure 2 illustrates this process. This funneling operation is inspired by the sensorless manipulation work of Erdmann and Mason [4], and allows the robot to drive itself arbitrarily close to a convex vertex of the environment, in spite of motion errors.

Our prior work presented an algorithm that can determine, given two convex environment vertices, whether it is possible to perform a guaranteed transition between those two vertices using this corner-finding technique. The algorithm then generates a complete navigation plan by searching for a path through a directed graph whose nodes are the convex vertices of the environment, and whose edges connect pairs of vertices between which a corner-finding transition exists.

Figure 3 shows an example of a plan generated by this approach. We observed, however, that because the algorithm can only generate plans that travel point-to-point between convex vertices, it generally fails for problems that require the robot to traverse certain environment features, such as T-junctions and long corridors. The new contributions of this paper are (1) a massive expansion to the usable nodes in the underlying directed graph to enable traversal of these kinds of features, (2) the consequent geometric algorithms for computing the edges in this expanded graph, and (3) a heuristic search procedure for exploring this graph efficiently.

IV. 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. 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, notice that convex vertices are formed whenever the two incident edges of a vertex form an interior angle less than or equal to π radians.

The robot is equipped with a compass and a contact sensor, but no other sensors. Note specifically that the robot has no clock nor any method of odometry, and consequently cannot measure the distances it moves. Using its compass, the robot can orient itself in a desired direction relative to a global reference frame, but because of noise in the sensor, this rotation is subject to potentially large, bounded error. Using its contact sensor, can translate in this direction until it reaches the boundary of the environment.

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

1) The *state space* $X = W$ is simply the robot's environment. Because we encapsulate the robot's use of its compass as part of the actions, we need not record the robot's orientation as part of the state.
2) The *action space* $U \in [0, 2\pi)$ is the set of planar angles. To execute an action $u \in U$, the robot orients itself in direction u, subject to the error described below, then moves forward in this direction until it reaches the environment boundary.
3) Time proceeds in a series of *stages*, numbered $k = 1, 2, 3, \ldots$. In each stage, the robot chooses and completes a single action. At stage k, the robot's state is denoted x_k and its action is denoted u_k.
4) Rotation errors are modeled as interference by an imaginary adversary called *nature*. In each stage, nature chooses a *nature action* $\theta_k \in \Theta$. Nature's action space Θ = ($-\theta_{max}$, $+\theta_{max}$) 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. 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 actions, so that 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}:

$$x_{k+1} = f(x_k, u_k, \theta_k). \tag{1}$$

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 $u_k + \theta_k$.

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

$$||x_G - x_{n+1}|| < \delta \tag{2}$$

for all possible nature action sequences $\theta_1, \ldots, \theta_n \in \Theta$. 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.

V. Algorithm description

In this section we describe an algorithm to solve navigation problems of the form given in Section IV. From the robot's environment, we create a graph with nodes representing sets of states and edges indicating the existence of transitions between such sets. Each edge is labeled with a sequence of actions $u_i, \ldots, u_{i+K}$ such that the robot will be guaranteed to make a transition between its incident nodes. We use two classes of nodes: *point nodes* representing convex vertices, and *segment nodes* representing positional uncertainty along some environment edge. A solution is a path in the graph between the point nodes representing $\{x_S\}$ and $\{x_G\}$. The bulk of the computation time is spent discovering the nodes and edges of the graph.

We describe a segment S by its endpoints $\mathrm{src}[S]$ and $\mathrm{tar}[S]$. By convention, we maintain the property that a counterclockwise rotation of the vector $\mathrm{tar}[S] - \mathrm{src}[S]$ is into the free space of W. For a (possibly degenerate) segment S and an action u, 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. See Figure 4. We call an action *safe* from a segment S if the far boundary of $e(S, u, \theta_{max})$ lies on a single edge of W. An interval of actions (d_1, d_2) is an *interval of safe actions* if every action in the interior of the interval is safe.

Several of the algorithms we describe utilize a function called ShootRay, that takes as input a starting point $p \in W$ and a direction, and returns the first boundary point contacted and the environment edge on which that point lies. This is a standard operation from computational geometry. It takes $O(\log n)$ time, in which n is the number of vertices in W [13].

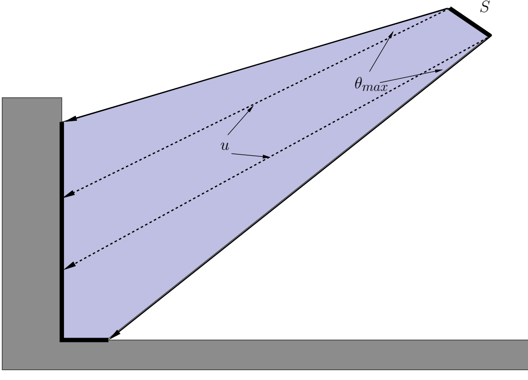

Fig. 4. An error cone whose far boundary spans two edges of W. As a result, the illustrated action u would not be in any interval of safe actions.

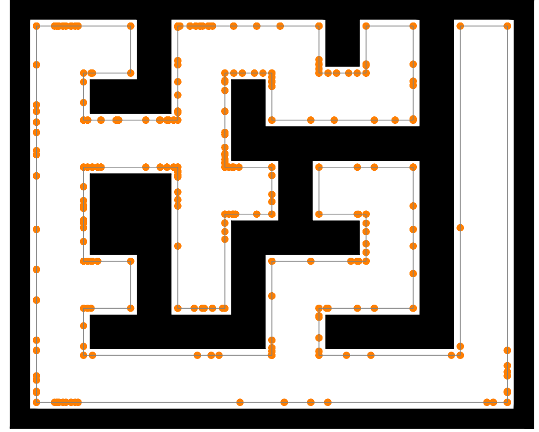

Fig. 5. An environment illustrating all the delimiting points generated by GenerateNodes algorithm.

A. Graph nodes

Using only the point nodes and corner-finding edges of our prior work, there are inherent limitations in the set of navigation problems we can solve. Specifically, for every intermediate transition between two states x_s and x_g, there must exist at least one safe action from $\{x_s\}$ that reaches one of the two edges incident to x_g. Certain environment features, such as T-junctions, generally do not satisfy this restriction.

To overcome this limitation, we introduce segment nodes into the graph to allow temporary position uncertainty beyond the bounds δ given in the definition of localization in Section IV. Each segment node χ corresponds to a closed line segment along a single boundary edge of W. When there is no possibility for confusion, we use χ to denote either a segment node or its underlying line segment. We also retain point nodes at each convex vertex of W.

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. See Section III for details. 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 θ_{max} radians away from that vertex.

Based on this observation, we generate a set of *delimiting points* that serve as the endpoints for our segment nodes. As shown in Algorithm 1, we use rays originating from pairs of mutually visible vertices, extending in directions rotated $\pm\theta_{max}$, to build the collection of delimiting points. Figure 5 illustrates these delimiting points. The algorithm then generates, in each environment edge, a segment node for each ordered pair of distinct delimiting points within that edge.

To bound the number of nodes generated, let n denote the number of vertices of the environment and let l_i denote the number of delimiting points on edge i. For any two vertices $v_i \neq v_j$, the algorithm will create at most two delimiting points originating from v_j, each of which lies on a single edge, so 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, the total number of nodes created is

$$\sum_{i=1}^{n}\left[(l_i^2 + l_i)/2\right] \leq \sum_{i=1}^{n}(l_i^2) \leq \left(\sum_{i=1}^{n} l_i\right)^2 \leq (2n^2)^2 \in O(n^4),$$

in which we use the fact that squaring is a convex, and therefore, superadditive function.

B. Graph edges

Recall that an edge $\langle \chi_s, \chi_g \rangle$ of the graph represents the existence of a sequence of actions which safely brings a robot from some node χ_s to node χ_g. We next present a collection of tests which are used in attempt to generate such sequences.

1) Point node to point node (with corner-finding): Our previous work [8] includes an algorithm that generates edges between pairs of point nodes using a "corner-finding" technique. See Figure III for details.

2) Point node to segment node (direct): To generate an edge that moves from a point node χ_s to a segment node χ_g in a single step, we need only to identify the intervals of safe actions from χ_s that reach χ_g. To do so, we perform a radial sweep about χ_s, testing between each pair of consecutive obstacle vertices reached by the sweep ray. Figure 6 shows an example in which there are two such intervals of safe actions and Algorithm 2 shows the details of the algorithm. The algorithm runs in $O(n \log n)$ time.

3) Segment node to segment node (direct): Now we consider how to decide whether the robot can move from a segment node χ_s to another segment node χ_g using a single action. This is a generalization of the problem in Section V-B.2. The intuition is that if all of the intervals of safe actions generated by Algorithm 2, across all of the (infinitely many) points along the continuum from $\mathrm{src}[\chi_s]$ to $\mathrm{tar}[\chi_s]$ as its starting point, have non-empty intersection, then such an edge exists.

It is trivially true that any interval of safe actions from the entire interval χ_s cannot contain any actions are not safe from both $\mathrm{src}[\chi_s]$ and $\mathrm{tar}[\chi_s]$. To determine this initial set of candidate actions, we execute Algorithm 2 twice, and

Algorithm 1 GENERATENODES(W, θ_{max})

1: $N \leftarrow$ empty set of delimiting points
2: **for all** vertices $i \in W$ **do**
3:    **for all** vertices $j \in W$ **do**
4:       **if** $i \neq j$
      and j is a reflex vertex
      and j is visible from i **then**
5:          $a \leftarrow \text{angle}(j - i)$
6:          $N \leftarrow N \cup \{\text{SHOOTRAY}(i, a + \theta_{max}, W)\}$
7:          $N \leftarrow N \cup \{\text{SHOOTRAY}(i, a - \theta_{max}, W)\}$
8:       **end if**
9:    **end for**
10: **end for**
11: $Q \leftarrow$ empty set of segments
12: **for all** edges $f \in W$ **do**
13:    $N' \leftarrow$ all points in N contained in f
14:    $N' \leftarrow N \cup \{\text{src}[f], \text{tar}[f]\}$
15:    $n \leftarrow \text{count}(N')$
16:    SORT(N') by the distance from src[f]
17:    **for** $p \leftarrow 1$ **to** n **do**
18:       **for** $q \leftarrow 1$ **to** $n - i$ **do**
19:          $Q \leftarrow Q \cup \{(\text{src}[N'[q]], \text{tar}[N'[q+p]])\}$
20:       **end for**
21:    **end for**
22: **end for**
23: **return** Q

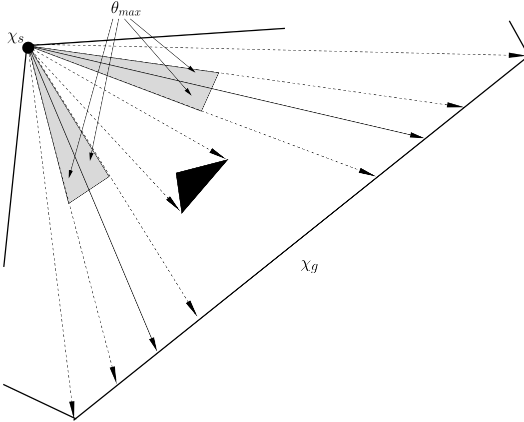

Fig. 6. A point node χ_s and a segment node χ_g for which Algorithm 2 would return true.

Algorithm 2 POINTTOSEGMENTTEST ($\chi_s, \chi_g, \theta_{max}, W$)

1: $V \leftarrow$ empty set of angles
2: **for all** vertices $v \in W$ **do**
3:    add $\text{angle}(v - \chi_s)$ to V
4: **end for**
5: SORT(V) counterclockwise 0 to 2π
6: **for** $i \leftarrow 1$ **to** count(V) **do**
7:    $a \leftarrow$ angle_bisector($V[i]$, $V[i]$.next)
8:    **if** $2\theta_{max} <$ angle($V[i]$.next $- V[i]$)
   and ray from s in direction $V[i]$ intersects g
   and ray from s in direction $V[i]$.next intersects g
   and SHOOTRAY(s, a, W) returns a point on g **then**
9:       **return true**
10:    **end if**
11: **end for**
12: **return false**

Algorithm 3 SEGMENTTOSEGMENTTEST($\chi_s, \chi_g, \theta_{max}, W$)

1: $P_s \leftarrow$ PTOS_SAFEINTERVALS(src[χ_s], χ_g, θ_{max}, W)
2: $P_t \leftarrow$ PTOS_SAFEINTERVALS (tar[χ_s], χ_g, θ_{max}, W)
3: $P \leftarrow P_s \cap P_t$
4: **for all** intervals $(d_1, d_2) \in P$ **do**
5:    **if** $2\theta_{max} < |d_2 - d_1|$ **then**
6:       $q \leftarrow$ quadrilateral with vertices src[χ_s], tar[χ_s], SHOOTRAY(src[χ_s]$, d_2, W$), and SHOOTRAY(tar[χ_s]$, d_1, W$)
7:       **for all** vertices $v \in W$ **do**
8:          **if** q contains v **then**
9:             **goto** line 14
10:          **end if**
11:       **end for**
12:       **return true**
13:    **end if**
14: **end for**
15: **return false**

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 χ_s.

To accomplish this, we consider the maximal intervals of actions safe from both src[χ_s] and tar[χ_s] in turn. For each such interval (d_1, d_2), we form a quadrilateral for which two vertices are src[χ_s] and tar[χ_s], and the remaining two vertices are the results of SHOOTRAY(src[χ_s]$, d_2, W$) and SHOOTRAY(tar[χ_s]$, d_1, W$). If any obstacle vertices are within this quadrilateral, then the entire interval can be discarded as unsafe. Figure 7 shows an example and Algorithm 3 shows the details.

The algorithm's two calls to Algorithm 2 each take $O(n \log n)$ time. To provide a maximum number of intervals constructed by the loops on lines 3 and 4 of Algorithm 3, consider a very contrived environment wherein all vertices $v \in W$ except the four vertices of χ_s and χ_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 at most $4(n-4)/3$ intervals generated between the two nodes. The test to determine whether a vertex lies inside the quadrilateral takes constant time, therefore a maximum number of intervals in $O(n)$ leads to a total run time in $O(n^2)$.

4) Segment node to point node (with corner-finding): To build edges that reach point nodes from segment nodes, we use a variation on the corner-finding technique referenced in Section V-B.1. The key extension we need is a function called CORNERFINDINGTARGET that returns the

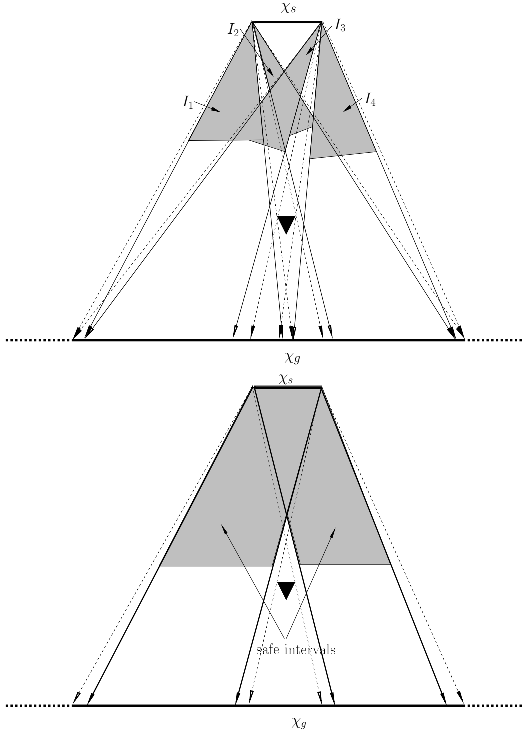

Fig. 7. [top] An example segment-to-segment test, in which there are two intervals I_1 and I_2 of safe actions from $\mathrm{src}[\chi_s]$ and two such intervals I_3 and I_4 from $\mathrm{tar}[\chi_s]$. [bottom] The final result includes $I_1 \cap I_3$ and $I_2 \cap I_4$. A third interval, $I_1 \cap I_4$ is safe from both endpoints, but is correctly rejected by the algorithm because its quadrilateral contains an obstacle.

Algorithm 4 SEGMENTTOPOINTTEST$(\chi_s, \chi_g, \theta_{max}, W)$

1: $(S_1, S_2) \leftarrow$ CORNERFINDINGTARGET$(\chi_g, \theta_{max}, W)$
2: **for all** $S \in \{S_1, S_2\}$ **do**
3: **if** SEGMENTTOSEGMENTTEST$(\chi_s, S, \theta_{max}, W)$ **then**
4: **return true**
5: **end if**
6: **end for**
7: **return false**

largest segments from which the corner-finding algorithm is guaranteed to succeed. There are always two such segments, one for each of the edges incident to χ_g. Details on how to compute such segments in $O(n)$ time appear in our prior work [8]. Using these segments, the edge test becomes a simple application of Algorithm 3 to these two candidates. Refer to Algorithm 4, whose run time $O(n^2)$ is dominated by the call to Algorithm 3.

5) Point node to point node with oscillation and corner-finding: A weakness in our approach to segment node generation is apparent in environments containing long edges with no environment vertices in between. An example of this structure is a long "hallway," in which there are 4 vertices at the ends and none in the middle. In this scenario, Algorithm 1 would generate segment nodes nearly as long as the hallway itself. An example is visible at the far right side of Figure 5. It is extremely difficult to generate outgoing edges from such

Algorithm 5 LONGHALLTEST$(\chi_s, \chi_g, u, \theta_{max}, W, l)$

1: $u_1 \leftarrow u + \theta_{max}$
2: $u_2 \leftarrow u - \theta_{max}$
3: $p_1 \leftarrow \chi_s$
4: $p_2 \leftarrow \chi_s$
5: $e_1 \leftarrow$ **nil**
6: $e_2 \leftarrow$ **nil**
7: $i \leftarrow 0$
8: **while** $e_1 = e_2$ and $i < l$ **do**
9: $p'_1 \leftarrow p_1$
10: $p'_2 \leftarrow p_2$
11: $\langle p_1, e_1 \rangle \leftarrow$ SHOOTRAY(p_1, u_1, w)
12: $\langle p_2, e_2 \rangle \leftarrow$ SHOOTRAY(p_2, u_2, w)
13: $q \leftarrow$ quadrilateral formed by p_1, p_2, p'_1, and p'_2
14: **for all** vertices $v \in W$ **do**
15: **if** q contains v **then**
16: **goto** 28
17: **end if**
18: **end for**
19: **if** $||(\mathrm{src}[e_1] - \chi_s)|| < ||(\mathrm{tar}[e_1] - \chi_s)||$ **then**
20: $u_1 \leftarrow \mathrm{angle}(\mathrm{tar}[e_1] - \mathrm{src}[e_1]) - \frac{\pi}{4} + \theta_{max}$
21: $u_2 \leftarrow \mathrm{angle}(\mathrm{tar}[e_1] - \mathrm{src}[e_1]) - \frac{\pi}{4} - \theta_{max}$
22: **else**
23: $u_1 \leftarrow \mathrm{angle}(\mathrm{src}[e_1] - \mathrm{tar}[e_1]) + \frac{\pi}{4} - \theta_{max}$
24: $u_2 \leftarrow \mathrm{angle}(\mathrm{src}[e_1] - \mathrm{tar}[e_1]) + \frac{\pi}{4} + \theta_{max}$
25: **end if**
26: $i \leftarrow i + 1$
27: **end while**
28: **return** SEGMENTTOPOINTTEST$(\langle p'_1, p'_2 \rangle, \chi_g, \theta_{max}, W)$

nodes using the techniques described above.

To address this shortcoming, we provide an algorithm whereby the robot attempts to make progress by alternating motions between the hallway walls. The algorithm chooses two initial action, offset from $\mathrm{angle}(\mathrm{src}[\chi_g] - \mathrm{src}[\chi_s])$ by $\pi/4$ in either direction and executes Algorithm 5 using each of these initial actions as the input u.

As described in Sections V-B.2 and V-B.3, we can calculate the forward projection of a robot's possible states from a single point or segment under a given action. 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 projection. This process continues until either (1) one of the 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 limit of l iterations. We provide this limit as it is not obvious that the algorithm will halt. After this process is complete, we invoke SEGMENTTOPOINTTEST to attempt to reach χ_g. This algorithm's run time is clearly $O(n^2)$, dominated by the call to Algorithm 4.

C. Searching for connections

Sections V-A and V-B describe a graph with as many as $O(n^4)$ nodes and $O(n^8)$ edges, each of which requires

several tests to include or discard. As a result, a brute force approach that performs the edge tests on all node pairs would be infeasible. Instead, we pre-process the environment and provide a heuristic to prefer pairs of nodes that are more likely to result in an edge between and also to be useful for navigation between a given starting node and ending node.

To compare two given node pairs $(\chi_{s,1}, \chi_{g,1})$ and $(\chi_{s,2}, \chi_{g,2})$, we apply a sequence of tests as listed below. We apply these tests as a sequence of "tiebreakers," 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 $\chi_{g,i}$ is a point node.
2) Prefer the pair for which the final node $\chi_{g,i}$ is closer, in geodesic distance, to the global goal x_G.
3) Prefer the pair for which the final node $\chi_{g,i}$ is smaller.

4,5) Repeat tests 2 and 3 for the initial nodes $\chi_{s,i}$.

The intuition 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. We also implement a filtering technique in which node pairs that are separated by three or more "turns" in the shortest path between them are discarded outright. Such node pairs are unlikely to be connected by any of our edge test algorithms.

The full planner uses a priority queue of node pairs, ordered by this heuristic. It also maintains a *connected set* C of nodes χ for which a plan reaching χ from x_S is known, and an *unconnected set* U containing all other nodes. The connected set C is initialized to contain only x_S and the priority queue is seeded with all pairs originating at x_S. In each iteration, a node pair is extracted from the queue and we execute each of the applicable edge tests described above to attempt to connect the nodes to each other. If any of edge tests succeed, we remove χ_g from U, add it to C, and insert the node pairs $\{\chi_g\} \times U$ into the queue. If x_G is inserted into C, then a plan is complete, and the action sequence can be extracted from the sequence of edges connecting x_S to x_G. If there is no way for a connection to be made between the start and goal nodes using the methods we've presented above, the algorithm will terminate with an empty queue, having attempted all possible pairs.

To speed the heuristic calculations, we preprocess the environment, by the construction of a visibility graph on all vertices of the environment, along with all-pairs shortest path data for this visibility graph. We use this structure to calculate the shortest path distance and the number of turns between each pair of environment vertices. In the case of point nodes, geodesic distance and turns are calculated directly. In the case of segment nodes, we overestimate the distance by using the environment edge on which the node lies. Specifically, for distances involving a segment node χ, we use the mean of two distances: (1) the distance from $\mathrm{src}[\chi]$ to the closest environment vertex of the segment on which χ lies, and (2) the distance from $\mathrm{tar}[\chi]$ to the closest environment vertex of the segment on which χ lies. In the case of a small segment node near its edge's endpoint, options (1) and (2) may be the same.

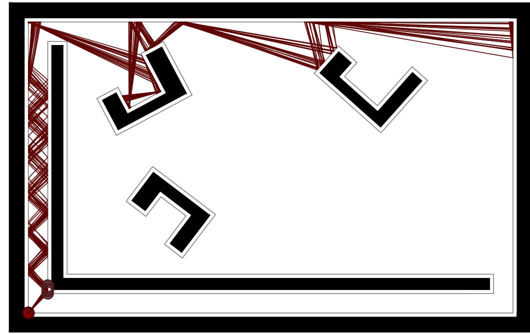

Fig. 8. A plan generated by our algorithm in a non-rectilinear environment. The plan is simulated 20 times and shown in parallel. The different paths are caused by interference up to $\pm\theta_{max} = \frac{\pi}{36}$.

VI. Implementation and Experiments

A. Simulated implementation

We explored the performance of our planner by implementing it in C++, using CGAL [2] as a geometry engine modeling the robot and environment. Figures 1, 8, and 9 are three of the environments on which we performed experiments. Figure 8 is a non-rectilinear environment with 42 vertices, divided by an obstacle inducing two very long halls. Figure 9 is an environment containing 172 vertices and several instances of long halls and T-junctions. Figure 1 is a rectilinear environment with 44 vertices used to collect data on the performance of several naïve approaches to the edge search.

The experiment illustrated in Figure 8 was run with $\theta_{max} = \frac{\pi}{36}$. The effect of uncertainty is evident in the cones created by the robot's different paths. This environment offers unique challenges due to the long halls and the obstacles set between the convex vertices of the graph. Our planner required 364 edges and 114,847 edge connection attempts to generate the solution shown. The solution to problem presented in Figure 9 used $\theta_{max} = \frac{\pi}{72}$ and required 193 edges and 24,004 connection attempts.

To evaluate our heuristic, we solved the planning problem depicted in Figure 1 using our heuristic method, and compared its performance to similar planners that use a random distribution, a stack, and a queue. The following table details the results, which confirm the success of our approach. It is worth noting the final plan found by each method were identical.

Heuristic comparison:

	heuristic	random	stack	queue
edges found	13	493	711	353
connection attempts	1,201	198,736	167,590	167,590

B. Physical implementation

We have implemented the algorithm on an iRobot Create, as shown in Figure 10. Because the Create platform does not have a compass sensor, it was necessary to simulate the compass via the Create's IR range sensor and encoders. This was done via noting the direction of each wall along which the robot ends its translation. Once adjacent to that wall, the

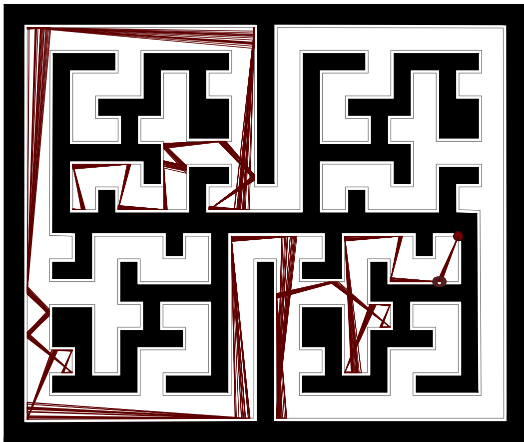

Fig. 9. A plan generated by our algorithm and simulated 20 times, each different due to interference in rotation up to $\pm\theta_{max}$. In this experiment, $\theta_{max} = \frac{\pi}{72}$.

Fig. 10. An iRobot Create executing a plan generated by our planner.

robot uses its range sensor to rotate to a pose approximately parallel to the wall, such that its right side is against the wall. We then used the robot's encoders to rotate a necessary offset for the proper pose facing. This technique allowed us to circumvent the accumulation of error normally seen in pure dead-reckoning or encoder measurement. While the simulated compass technique held, the Create performed as in simulation.

VII. Discussion and conclusion

We have presented a planning algorithm which generates plans that a robot having only a compass, map, and contact sensor can use to navigate non-trivial environments containing obstacles. The planner considers the bound on error in the robot's actions and chooses plans to move the robot between sets of states in the environment, driving it from a singleton initial set to a singleton goal set through the use of specialized high-level actions. The planner represents the sets as nodes in a graph and the existence of an edge guarantees safe transitions between nodes. In our previous work, our algorithm sought to fill the graph, then generate plans via a simple breadth-first search. The enhanced graph in this work is too large and complex to calculate explicitly, so instead we use a heuristic to focus the search for edges and terminate the algorithm once it has exhausted all possible pairs or successfully generated a solution. We have also presented data to support the power of our heuristic and several non-trivial environments through which our planner succeeded in navigating.

We are also working to prove the hardness of a complete solution to this problem. In our current work, we have several limiting restrictions in that we only consider one-step plans between nodes, generate a finite discretized selection of segment nodes, and limit segment nodes to subsets of environment edges. It appears likely that a complete solution will consider plans of length n, will provide some mechanism to use exactly the segment nodes necessary for those plans, and will not restrict segment nodes to a single environment edge.

Acknowledgments

We gratefully acknowledge support from NSF (IIS-0953503) and DARPA (N10AP20015).

References

[1] M. Blum and D. Kozen, "On the power of the compass (or, why mazes are easier to search than graphs)," in *Proc. IEEE Symposium on Foundations of Computer Science*, 1978, pp. 132–142.

[2] "CGAL, Computational Geometry Algorithms Library," http://www.cgal.org.

[3] B. R. Donald, "On information invariants in robotics," *Artificial Intelligence*, vol. 72, pp. 217–304, 1995.

[4] M. Erdmann and M. T. Mason, "An exploration of sensorless manipulation," *IEEE Transactions on Robotics and Automation*, vol. 4, no. 4, pp. 369–379, Aug. 1988.

[5] M. A. Erdmann, "Understanding action and sensing by designing action-based sensors," *International Journal of Robotics Research*, vol. 14, no. 5, pp. 483–509, 1995.

[6] L. Erickson, J. Knuth, J. M. O'Kane, and S. M. LaValle, "Probabilistic localization with a blind robot," in *Proc. IEEE International Conference on Robotics and Automation*, 2008.

[7] A. Lazanas and J. C. Latombe, "Landmark-based robot navigation," in *Proc. National Conference on Artificial Intelligence (AAAI)*, 1992.

[8] J. S. Lewis and J. M. O'Kane, "Guaranteed navigation with an unreliable blind robot," in *Proc. IEEE International Conference on Robotics and Automation*, 2010.

[9] T. Lozano-Pérez, M. T. Mason, and R. H. Taylor, "Automatic synthesis of fine-motion strategies for robots," *International Journal of Robotics Research*, vol. 3, no. 1, pp. 3–24, 1984.

[10] J. M. O'Kane and S. M. LaValle, "Localization with limited sensing," *IEEE Transactions on Robotics*, vol. 23, pp. 704–716, Aug. 2007.

[11] ——, "On comparing the power of robots," *International Journal of Robotics Research*, vol. 27, no. 1, pp. 5–23, Jan. 2008.

[12] N. Roy and S. Thrun, "Coastal navigation with mobile robots," in *Advances in Neural Processing Systems*, 1999, pp. 1043–1049.

[13] L. Szirmay-Kalos and G. Marton, "Worst-case versus average case complexity of ray-shooting," *Computing*, vol. 61(2), no. 2, pp. 103–131, 1998.

[14] B. Tovar, L. Guilamo, and S. M. LaValle, "Gap Navigation Trees: Minimal representation for visibility-based tasks," in *Proc. Workshop on the Algorithmic Foundations of Robotics*, 2004.