Vision Based Robot Navigation

by

Daniel R. Roth

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2004

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2004

Certified by .....
Leslie P. Kaelbling
Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Vision Based Robot Navigation

by

Daniel R. Roth

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis we propose a vision-based robot navigation system that constructs a high level topological representation of the world. A robot using this system learns to recognize rooms and spaces by building a hidden Markov model of the environment. Motion planning is performed by doing bidirectional heuristic search with a discrete set of actions that account for the robot's nonholonomic constraints. The intent of this project is to create a system that allows a robot to be able to explore and to navigate in a wide variety of environments in a way that facilitates goal-oriented tasks.

Thesis Supervisor: Leslie P. Kaelbling
Title: Professor

Acknowledgments

Many thanks to Kevin Murphy, and Georgios Theocharous for their excellent mentorship. Also thanks to Michael Ross, Luke Zettlemoyer, Natalia Hernandez, Sarah Finney and Sam Davies for their help and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A robot navigation system is a system that allows an autonomous robot to move throughout its environment under constraints, such as avoiding obstacles, estimating location, building an accurate map or attaining some goal location. In this thesis we present the groundwork for a vision-based topological navigation system.

Many current robot navigation systems make restrictive assumptions about the environment the robot will navigate in, such as assuming a two-dimensional world or assuming the world consists of hallways and lobbies. Vision provides a rich set of features that should allow a robot to navigate and map a larger variety of environments. Many robot navigation systems also focus on producing a detailed metric map of the world using expensive hardware, such as a laser scanner. After this map is built the robot then has to solve a complicated path-planning problem. An abstract topological representation simplifies the task of planning a path to a goal and does not require expensive, high accuracy range sensors.

Humans can readily demonstrate the ability to navigate visually without the supplement of metric information [9]. This ability has inspired the development of view-based robot navigation systems, which represent an environment as a set of snap shots and also a set of control sequences or algorithms for navigating between those snapshots. View-based approaches have been shown to be effective for both exploration and navigation tasks [2].

View-based approaches operate well in static environments but are inherently not

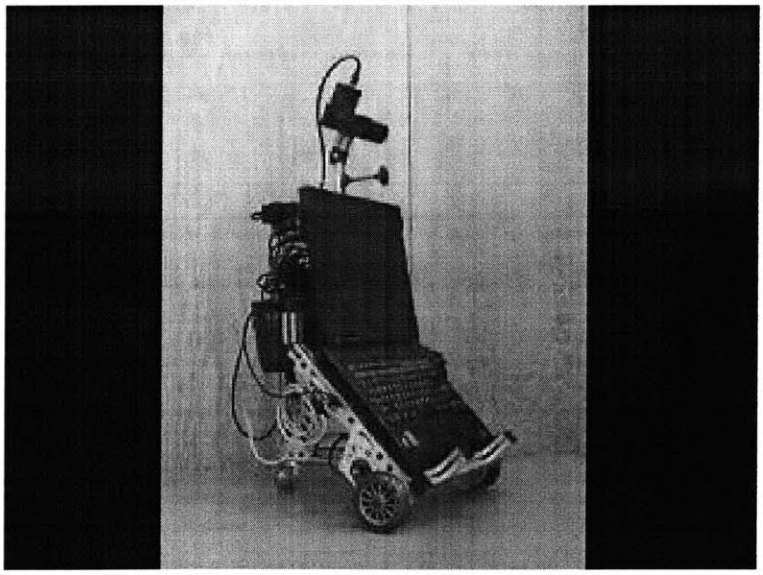

Figure 1-1: Gerry: A visually navigating robot.

robust to a dynamic environment. View-based approaches also scale poorly since the topological representation tends to be fine grained and therefore requires storing many views. Experiments on humans indicate that people do not represent space as a collection of views but instead reason about landmarks in an environment at a higher level of abstraction [8]. A more abstract model-based approach has the potential to deal with dynamic environments and to compress the information stored in collections of views.

The navigation system presented in this thesis was implemented on an ER1 robot from Evolution Robotics (see Figure 1-1). The software was developed using the ERSP robotic development platform from the same company. The ER1 robot runs off of a standard laptop with USB peripherals including infrared distance sensors and a gripper arm (not used for this system). The robot has a differential drive mechanism and a third passive caster wheel. We added a stereo camera to the robot to perform the localization functions and to also provide range data for obstacle avoidance.

Chapter 2

Localization

Any robot navigation system that intends to tell a robot how to move will need to know where the robot is. Localization is the process of estimating the robot's state within its environment and is one of the primary pieces of any robot navigation system. Gerry's localization system operates at the level of abstraction of a room or space, which makes the localization system extremely reliable. The input to the localization system is also purely visual, which makes the system ammendable to a variety of environments.

Gerry's navigation system relies on a vision-based localization system developed by Murphy, Torralba and Freeman [12]. The robot determines its location by calculating what room or location the images from its video camera appear most similar to. The system models the different locations in its environment as states in a hidden Markov model (HMM). The robot observes the environment using one of its video cameras and then calculates the probability that it is in any given state given all past observations using its HMM. The topology of the environment is accurately represented in the HMM's state transition probability matrix which allows the robot improve its localization estimate by considering the connectivity of the space.

2.1 Image Processing

The localization system first performs a series of image processing steps on the images to extract a small set of features that are reasonably invariant to illumination changes and affine transformations. The image representation used by the system is designed to capture global texture properties while keeping some spatial information.

The image is first converted to a grayscale image and is then decomposed using a steerable pyramid filter bank with 6 orientations and 4 scales on the intensity values of the image. An excellent tutorial on steerable pyramids with implementation code can be found in [11] and on Eero Simoncelli's web site.[1] Steerable pyramids are linear, multi-scale, multi-orientation image decompositions. The basic operators are directional derivatives performed at multiple scales and orientations. The result is an image format that is translation and rotation invariant. The representation is overcomplete by a factor of $4k/3$ where k is the number of subbands, however its properties make it a reasonable choice for our localization task. Similar localization results using Gabor filters are reported in [12] although we did not peform any experiments in this direction.

In order to reduce the dimensionality of the image representation, but still preserving global spatial and textural information, each subband of the filter output is divided into an M by M grid and only the mean value of the magnitude of each cell is kept (we used $M = 4$ as was done in [12]). The result is a 384 dimensional (6 orientations x 4 scales x 16 grid cells) feature vector. The feature vector is further reduced in dimensionality to 80 dimensions using principal component analysis.

2.2 Modeling the Environment

We model the locations in the environment and their interconnectivity using a hidden Markov model (HMM). An excellent tutorial on hidden Markov models and their applications can be found in [10] from which relevant topics are discussed here.

A hidden Markov model can be used to describe an observation sequence $O =$

[1] http://www.cis.upenn.edu/ eero/steerpyr.html

$O_1 O_2 \ldots O_T$ taken over a set of T timesteps. An HMM consists of a set of N states denoted as $S = \{S_1, S_2, \ldots, S_N\}$ and a state transition probability distribution $A = \{a_{ij}\}$ where

$$a_{ij} = P[q_{t+1} = S_j | q_t = Si], \quad 1 \leq i, j \leq N$$

An HMM also defines a set of probability distributions $B = \{b_1, b_2, \ldots, b_N\}$, where b_j defines the probability distribution over observations made in state S_j. The current state is denoted q_t. If the observations are continuous vectors, then the observation probability densities can be represented as a finite mixture of the form:

$$b_j(\boldsymbol{O}) = \sum_{m=1}^{M} c_{jm} \mathfrak{N}[\boldsymbol{O}, \boldsymbol{\mu}_{jm}, \boldsymbol{\Sigma}_{jm}], \quad 1 \leq j \leq N$$

where c_{jm} is the mixture component for the mth mixture in state j and $\mathfrak{N}$ can be any log-concave or elliptically symmetric density, although it is typically Gaussian. The mixture components satisfy the constraints

$$\sum_{m=1}^{M} c_{jm} = 1, \quad 1 \leq j \leq N$$

$$c_{jm} \geq 0, \quad 1 \leq j \leq N, \; 1 \leq m \leq M$$

so that the pdfs are properly normalized.

Once the HMM has been constructed, the probability of being in state q at time t is given by the recursive equation:

$$\begin{aligned} P(Q_t = q | O = O_{1:t}) &\propto p(O_t | Q_t = q) P(Q_t = q | O = O_{1:t-1}) \\ &= p(O_t | Q_t = q) \sum_{q'} A(q', q) P(Q_{t-1} = q' | O = O_{1:t-1}) \end{aligned}$$

where $p(O_t | Q_t = q)$ is the likelihood of an observation given the state, and $A(q', q)$ is the transition probability from state q' to state q.

The localization system is trained by recording sequence of images from the robot as it initially explores the environment and then labeling the images with the name

of the location where the image was taken. These labels could also be determined automatically using expectation maximization, although we did not attempt this.

The observation probability densities are estimated by randomly selecting K prototype observations from each location and then setting the mixture component variance to be a constant, σ_p, where p stands for "prototype." The parameters K and σ_p were set to 100 and 0.05 respectively using cross-validation as explained in [12]. Each mixture component is weighted equally, resulting in what is essentially a Parzen window density estimator.

The state transition probability matrix is estimated by counting transitions between states in the training data. A Dirichlet smoothing prior is added to the count matrix so that zero likelihoods are not assigned to transitions which do not appear in the training set. The prior probabilities are calculated by assuming that the robot is equally likely to start in any location.

Because the space of the feature vectors is significantly larger than the number of locations, the observation likelihoods tend to overwhelm the transition probabilities. To compensate for this, we rescale the likelihood terms to be:

$$b_q = \frac{p(O_t|Q_t = q)^\gamma}{\sum_{q'} p(O_t|Q_t = q')^\gamma}$$

where γ is set by cross validation as described in [12]. This rescaling improves performance at the cost of increasing the latency of the localization system. A higher latency means that there is a longer time lapse bewteen when Gerry enters a location and when that location is recognized. In practice, the system latency remained less than a few seconds.

2.3 Robot Implementation and Performance

We implemented the localization system on Gerry and trained the system on the rooms and hallways surrounding our lab. A map of the seven rooms trained on can be seen in Figure 2-1. We drove the robot around our lab and recorded four sequences of images from the left imager of the robot's stereovision camera. The images were

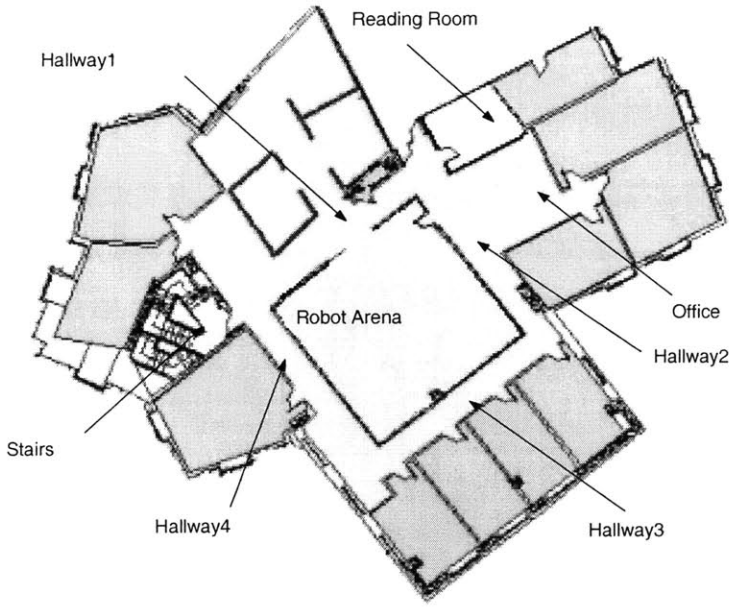

Figure 2-1: Map of the room locations used to train the localization system for testing.

recorded at roughly 5 Hz for a total of 1676 images in the first sequence, 776 in the second, 617 in the third, and 332 in the forth. Each sequence visits all locations and all transitions between locations. The number of images per sequence decreases only because the robot driver became increasingly efficient at moving the robot through the locations.

Figure 2-2 shows the performance of the localization system by plotting the belief state of the localization system for each frame in the testing set. The plotted line shows ground truth and the gray circles indicate a belief. A large dark circle indicates high confidence and a smaller circle indicates ambiguity and confusion between states. The system is correct with high confidence for almost every frame in the training set except for a brief error around frame 360. This error is likely to labeling ambiguity in the transitions between the states Hallway1, Hallway2, and the Reading Room.

The system runs in real time on our robot. The HMM is updated at a rate of approximately 5Hz on a 1000MHz Intel processor with 256Mb of memory. We only tested an environment with eight rooms, but in tests reported in [12] up to 63 possible locations are recognized with similar performance.

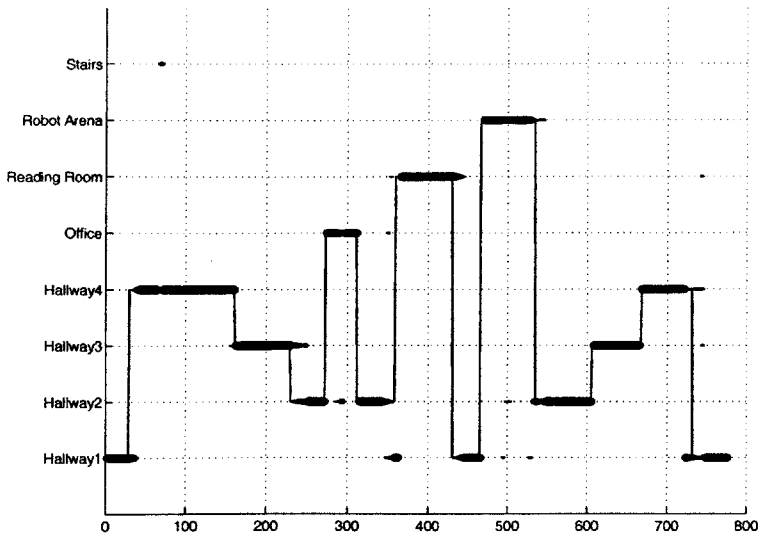

Figure 2-2: Localization performance over seven states.

We tested the system using different percentages of the training data, and performance was virtually unaltered even at 12.5% of the training data. Precision-recall curves generated through cross-validation are in Figure 2-3 for different percentages of the training data. We generated these curves by subsampling the training sequence. The same sequences were used for training and testing for all tests. The precision-recall curves show the percentage of the testing set labeled by the system correctly as the confidence threshold of the system is decreased. We can see that the topological constraints imposed by the HMM improve the performance of the system, although the mixtures of Gaussians by themselves are already very accurate in this environment.

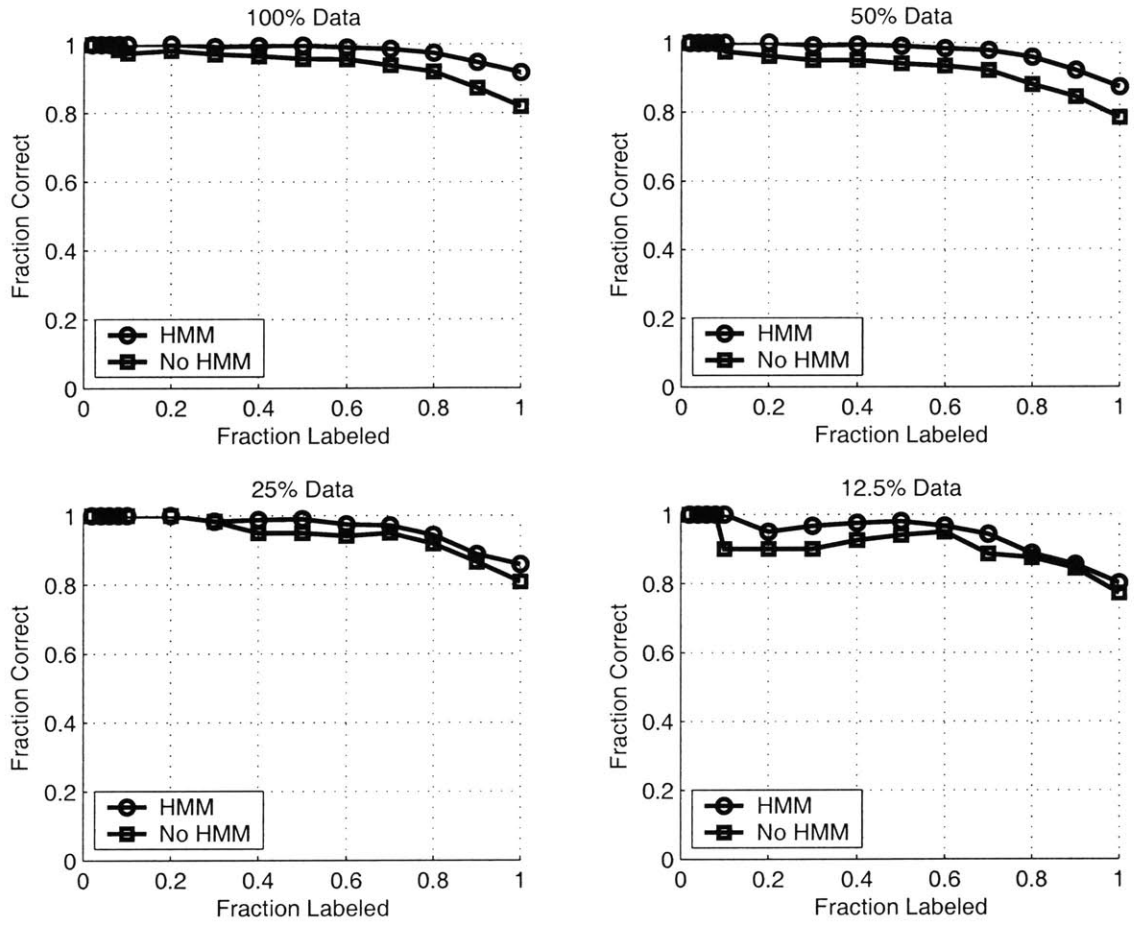

Figure 2-3: Precision-recall curves showing the performance of the localization system as the amount of training data is decreased.

Chapter 3

Motion Planning

Techniques for performing motion-planning with obstacle avoidance range in complexity from simple behavior-based approaches to complex global path-planning schemes. The simplest approaches are reflexive in nature. Some examples include backing up and spinning in a random direction if the robot bumps into something, or turning left if an obstacle is detected on the right, and turning right if an obstacle is detected on the left. These reactive approaches tend to be very robust and adaptive to unstructured environments, but also tend to be inefficient and non-optimal.

On the other end of the spectrum, generalized motion-planning solutions attempt to find optimal paths through a carefully measured environment. Motion planning can find optimal paths through complex environments, but also tends to be brittle and to not scale well to large environments. For Gerry's obstacle avoidance system we tried to find an optimal compromise between complexity with efficiency, and simplicity with robustness. Gerry performs local motion planning using bidirectional heuristic search over a discrete action set in a grid world representation.

3.1 Mapping the World

Gerry uses an off-the-shelf stereovision camera (from Videre Design) to see obstacles in the world. The stereovision camera consists of two video cameras set in a fixed geometry. Software can then be used to calculate the disparity between the pixels in

the images from the two cameras and infer the three-dimensional geometry. We chose to use data from a stereovision camera because we required a vision-based system and because it allows Gerry to see obstacles in all three dimensions: length, width and height. This allows Gerry to see things such as tabletops, which might appear only as four thin posts to a laser range finder or other distance sensors.

At a regular time cycle, Gerry's path planning system receives a set of three-dimensional points from the stereovision camera that are then used to create an occupancy grid. An occupancy grid (also called an evidence grid) is a world representation which lays a two dimensional grid over the world and then labels grid cells as either obstructed or free. The robot can readily traverse a free cell but obstructed cells are impassable.

The occupancy grid is created by flattening the stereocamera data along the z-dimension into what would look like a floor plan. The points are binned in a memory array equal in size to the occupancy grid where an array element represents an occupancy grid cell. Only those points that are within the height of the robot and within a fixed square area surrounding the robot are considered when building the occupancy grid. A cell is considered obstructed if the bin count is above a given threshold that is set by the user. The value for the threshold will depend on the area covered by a grid cell and the amount of noise in the data.

The coordinate frame used to build the occupancy grid is a Cartesian coordinate system in the robot's frame of reference. By convention, the x-axis is in the direction that the robot is facing, the y-axis is to the robot's left and the z-axis points up. The robot's orientation, θ, is measured counter-clockwise from the x-axis. Since Gerry is a differential-drive robot, we set the origin of the robot's coordinate system to be the midpoint of the robot's drive-wheels. We choose to center the occupancy grid on the robot to maximize the chance that a goal point will remain within the bounds of the grid even as the robot turns. We consider all points outside of the grid to be obstructed.

We conducted tests on the stereocamera data and the motion-planning system in a simple world consisting of two boxes acting as obstacles. For these experiments,

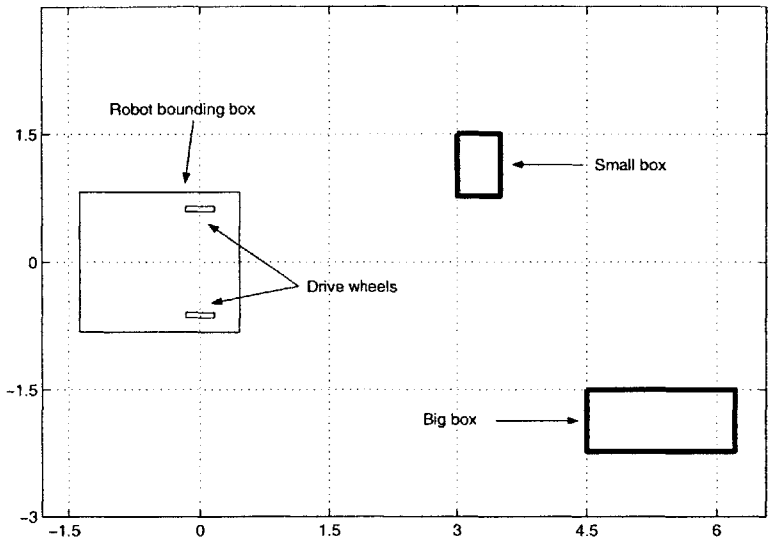

Figure 3-1: Test world setup for stereo and motion-planning experiments.

the robot and the boxes were positioned as shown in Figure 3-1. The small box is $6'' \times 8\frac{3}{4}'' \times 14''$ (15.2cm $\times$ 22.2cm $\times$ 35.6cm) and the large box is $20\frac{1}{2}'' \times 8\frac{3}{4}'' \times 15''$ (50.1cm $\times$ 22.2cm $\times$ 38.1cm). We positioned the boxes using the carpet on the floor, which is laid out in convenient $1\frac{1}{2}' \times 1\frac{1}{2}'$ squares. The midpoint of the robot's drive wheels is positioned as shown, and the robot is facing along the x-axis.

We modeled the robot's geometry using a rectangular bounding box with dimensions 56cm $\times$ 50cm $\times$ 78cm. The robot's stereocamera is mounted on its neck so it can see the tops of the boxes, but also tilted downward at a slight angle so that the bottom of the images see just above the front of the robot. The stereocamera lenses have a focal length of 4.8mm, which equates to a 90° horizontal field of view and a 73° vertical field of view.

3.2 Filtering the Stereo Data

The stereo data can be very noisy, which causes virtual clutter to appear in front of the robot. Figure 3-2 shows an example 64 $\times$ 64 occupancy grid constructed from the stereocamera data in the setup described in Section 3.1. The origin is located at the middle with the robot facing right. The first image highlights all of the bins in

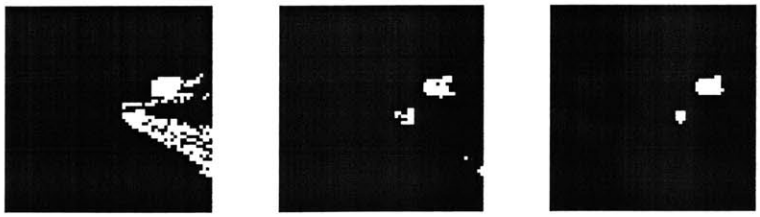

Figure 3-2: An occupancy grid constructed from the stereo camera data.

the occupancy grid that have values in them. As you can see, the data is somewhat messy. The second image is after applying a static threshold and the third image is after performing a median filtering. These operations are described below.

To determine the best threshold to use, we collected a set of 47 sample occupancy grids with the robot in the configuration shown in Figure 3-1. Then we constructed the Reciever Operating Characteristic (ROC) curves shown in Figure 3-3 using three different filtering techniques: time integration, median filtering, and combined. The curves have been truncated to highlight the interesting regions near the y-axis.

An ROC curve is a plot of the false positive rate versus the true positive rate for a binary classifier. The area underneath the ROC curve, called the discrimination value, identifies how well the the binary classifier would discriminate between two entities chosen at random from two different populations. We constructed these curves by gradually lowering the threshold and then counting the number of true positives and true negatives. Median filtering was performed over a 3×3 neighborhood and time integration was performed by summing over three time steps. The descrimination values were calculated using a trapizoidal approximation and are given in Table 3.1.

Median filtering has the effect of removing outliers by replacing pixels with the median of their surrounding pixels. This helps to cut back on noise in the stereo data

Filter	Discrimination
None	0.887
Time	0.907
Median	0.883
Both	0.893

Table 3.1: Discrimination values after applying different filtering techniques.

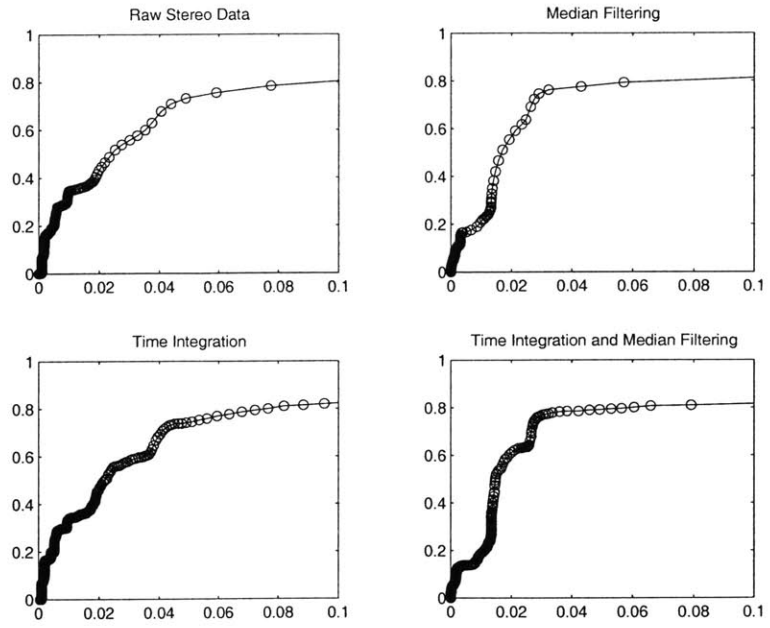

Figure 3-3: Stereo data threshold ROC curves for different filtering techniques.

and to stabilize edges. We can see the effects of the median filtering in the higher true positive rates that can be achieved at a much lower false positive rate. Median filtering has the unfortunate side effect, however, of making all obstacles that are one grid-cell wide invisible. This is evidenced by the lower discrimination value of 0.88 after performing median filtering.

Time integration allows the robot to see obstacles better because it can remember what it has seen before. Time integration results in a significantly higher discrimination value of 0.91. However, because the noise is not removed the false positive rate remains high. Combining time integration and then median filtering appears to have the benefits of both techniques. Using the ROC curve for time integration and median filtering combined, we selected a threshold of 21 which gives a false positive rate of 0.028 and a true positive rate of 0.753.

3.3 Calculating the Configuration Space

When performing path planning, it is generally easier to consider the robot's configuration space. Configuration space is a representation of all the valid states of the

robot. In our case, the robot's state consists of its position and orientation: (x, y, θ). A free point in configuration space represents a state where the robot is not colliding with any obstacles. By using the robot's configuration space, the motion-planning problem is reduced to finding a trajectory through configuration space from an initial state to a goal state along which all points are free and valid state transitions can be made between all consecutive pairs. Such a trajectory is said to be admissible. Not all trajectories through configuration space are necessarily admissible if there are constraints on the robot's kinematics. For example, Gerry cannot translate sideways.

In general, calculating the configuration space is a non-trivial problem. Efficient and optimal solutions exist for robots and obstacles that are polygonal [7]. An alternative is to represent the robot and the world as binary bitmaps and the then convolve the two [4]. The result is an approximation to the configuration space and can be computed quickly by performing the convolution in the frequency domain. The bitmap of the robot and the world can be transformed into the frequency domain by applying the fast Fourier transform to both bitmaps. Multiplying the corresponding elements in the two transformed bitmaps then performs the convolution. The result is transformed back into the spatial domain by using the inverse fast Fourier transform and then taking the magnitudes where are less than one.

If we wish to consider all points outside of the bounds of the occupancy grid as being obstructed, a simple trick is to draw a line around the border of the occupancy grid before performing the convolution [4]. If a more accurate approximation is needed then the resolution of the bitmap representations can be increased, although some fast Fourier transform implementations are most efficient when the dimensions of the input signal are a multiple of 2.

A single convolution is all that is needed to calculate the configuration space for a circular robot. If the robot is not circular, then the robot's orientation can be accounted for by calculating the configuration space for the robot at different discrete orientations. Again, if finer resolution is needed, then the number of orientations can be increased at the cost of computational time and space.

For Gerry, we calculate the configuration space at a resolution of 128×128 over

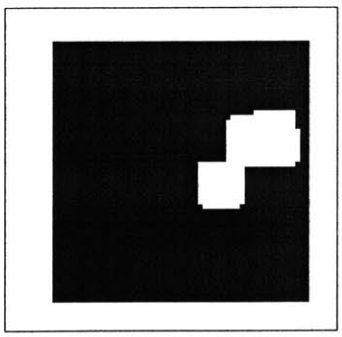

Figure 3-4: Configuration space calculated using binary bitmap representations and fast convolution.

16 evenly spaced orientations. Since the bitmap of the robot does not change, we can precompute the FFT's for the robot orientations offline which helps improve performance. The configuration space for the test world introduced in Section 3.1 is shown in Figure 3-4.

3.4 Robot Model

Gerry is a differential-drive robot with two parallel independently driven wheels and a passive caster wheel. A differential-drive robot has the advantage of being very simple to build and control, but tends to have difficulty going in straight lines. Differential-drive robots also are subject to nonholonomic constraints, which are constraints on the robot's kinematics that involve the configuration parameters and their derivatives, but are non-integrable. For example, the robot cannot translate sideways. Nonholonomic constraints complicate the motion-planning problem because they restrict the set of admissible trajectories in configuration space. The motion planner needs to be able to rule out these inadmissable trajectories.

For Gerry, we describe the kinematics of the robot's drive system using a simple bounded-velocity robot model, where the robot's left wheel velocity, v_l and the right wheel velocity, v_r are constrained such that $v_l, v_r \leq |v_{max}|$. We give no bounds on the acceleration of the wheels and we even allow discontinuities in the wheel velocities as described in Balkom and Mason [1]. Given that the distance between the robot

wheels is $2b$ we then have:

$$v = \frac{1}{2}(v_r + v_l)$$

$$\omega = \frac{1}{2b}(v_r - v_l)$$

and

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} cos(\theta) \\ sin(\theta) \\ 0 \end{pmatrix} v + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \omega.$$

At a given time instant, we can think of a differential drive robot as moving in a circular arc of radius:

$$r = \frac{b(v_r + v_l)}{2(v_r - v_l)}$$

where the sign of r is the same as the sign of θ. When $v_l = v_r$ we can consider the radius to be infinite and the robot moves in a straight line.

Time-optimal paths for bounded-velocity differential-drive robots in free space (spaces without obstacles) can be enumerated as combinations of pivots and straight line segments which are derived in [1]. There is no known algorithm for calculating time optimal paths in obstructed spaces, although a variety of techniques exist for planning reasonable paths with nonholonomic constraints for a variety of different robots [6]. Most of these techniques involve a three-phase procedure. The first phase is to compute a path while ignoring the nonholonomic constraints. The path is then broken up into subpaths that can be replaced by short admissible paths. Finally, the path is smoothed and optimized. An example of such a planner for a fixed, fully observable workspace can be found in [5].

Rather than incur the computational expense and complexity of smoothing an approximate path through configuration space, we instead choose to limit the action set of the search algorithm used by the motion planner to only construct admissible paths. In particular, we chose to only allow the robot to move forward, backward, pivot, and traverse 90° circular arcs. Doing so drastically simplifies the planner and the task of converting the path into a set of controls that can be executed.

3.5 Searching for a Path

We can find an unobstructed path through configuration space by discretizing the configuration space and then employing some form of search algorithm. Considering all possible actions and all possible states to which the robot can transition is unrealistic because the size of the search space would be too large or infinite. Instead we chose to represent the world as a set of discrete states and we select a subset of the robot's actions that correctly transition the robot between those states while not limiting the robot's capabilities too severely.

Methods for discretizing the configuration space take on two general forms: skeletonization and cell decomposition. Skeletonization reduces the configuration space to a one-dimensional space consisting of a network of connected curve segments through free space, collectively called a skeleton. Cell decompositions break the configuration space into adjacent chunks. For our system we chose a uniform grid cell decomposition at multiple discrete orientations for its flexibility and simplicity.

We create discrete states in the configuration space by laying a two-dimensional grid over the world and then considering only a discrete set of orientations. The discrete coordinates and orientations used for path planning need not correspond directly to the discretization used to calculate the configuration space. For example, it might be advantageous to search through a very simplified state space but then check whether an action is obstructed at a much higher resolution. For Gerry, we only search over actions that transition Gerry between states with cardinal orientations, but we check that these actions are possible by examining the configuration space over a finer set of 16 evenly spaced orientations. We can map between the states in the search space and the configuration space by computing continuous state values.

Search is the process of exploring sequences of actions to determine a sequence that leads to a desired goal state. An action performs a transition from one state to another. A sequence of actions from the initial state is called a search node. The search nodes form a search tree, where the fringe of the search is at the leaves. In robot motion-planning the goal state is some desired point in configuration space and the actions represent motions for the robot to perform. We approximate the

initial state and the goal state by converting them to the nearest discrete state in the discretization of the configurations space being used, and then search for a sequence of unobstructed actions that, when completed, end at the goal state.

The search algorithm is initialized by placing the goal state in a priority queue. The algorithm then proceeds by removing the first node on the priority queue and checking if the goal state has been reached. If it has not, the node is expanded into its neighboring nodes, which are in turn placed on the queue. The search continues until the goal state is found, the search space is exhausted, or some other stopping criterion occurs, such as a timeout.

Depending on the position of the robot, the position of the goal, and the positions of the obstacles in the world, there may not be a legal path from the robot's initial state to the goal. This condition can be detected by checking whether or not the search queue is empty when the search terminates, and it should be reported to the procedure's caller.

Different priority queue implementations yield different search behaviors. A priority queue that orders search nodes based on the cost:

$$f_{st}(n) = g(s, n) + h(n, t)$$

where $g(s, n)$ is the cost of the path from the start node s to a node n, and $h(n, t)$ is the estimated cost from a node n to the goal node t yields the efficient heuristic search algorithm called A*. A* returns an optimal path sequence if the heuristic used to estimate the cost to the goal is admissible,

$$h(n, t) \leq g^*(n, t)$$

or in other words, if the heuristic function h from node n to the goal node t is always an underestimate of the optimal cost $g^*(n, t)$ from n to t (the "$*$" stands for optimal or shortest). A commonly used admissible heuristic in motion planning is the straight line distance to the goal, although other heuristics are possible.

Search is computationally expensive, as it is generally exponential in the depth of

the search in both space and time. A perfect heuristic would change the running time to be linear, but perferct heuristics are difficult to come by. We can speed up the A* algorithm and reduce the size of the search problem to the size of the state space by eliminating loops from the search tree. In other words, we want to avoid considering the same actions from a given state multiple times, and also stop pursuing multiple action sequences that all lead to the same state. The search space for Gerry's actions is inherently loopy, since Gerry can drive forwards, backwards, and in circles. We can eliminate these loops by keeping the search tree in memory and only expanding a search node taken from the queue if its state has not been expanded before. The nodes that have already been expanded are sometimes called the closed list or the expanded list. The nodes still on the queue are called the open list, visited list, or the fringe.

Changing the search algorithm to only expand nodes that have not been expanded before changes the optimality condition on the search heuristic. The heuristic must also be consistent, which means it must obey the triangle inequality:

$$h(m,t) \leq g(m,n) + h(n,t)$$

for an intermediary node n. Fortunately, admissible heuristics that are not also consistent appear to be somewhat rare, and the straight line distance heuristic is both admissible and consistent.

Keeping the entire search tree in memory is expensive but feasible for reasonably sized local path planning problems. We can reduce the space cost by only keeping in memory one node per state, and then also checking whether a node to be put on the queue represents a state that has been put on the queue before. If a node representing the same state has already been put on the queue, then the cost of the node must be checked. If the cost of the node already on the queue is less than the cost of the new node, then the new node need not be put on the queue as well. Otherwise the cost of the node in memory should be updated with the smaller cost, and the new node should be put on the queue and the old node should be taken off. Removing the old node may not be practical depending on the priority queue implementation,

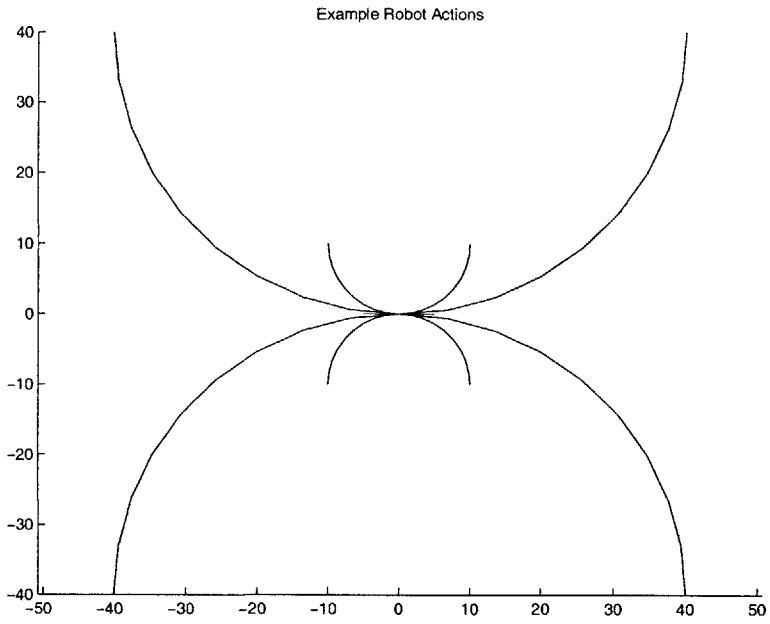

Figure 3-5: An example action set of size 12.

but leaving it on the queue does not effect the order of node expansion so we do not bother with this step in our implementation.

We limit Gerry to only move in 90° arcs with radii that are a multiple of the grid resolution. Allowing more general arcs would cause the size of the search space to explode. An alternative would be to tessellate the world with equilateral triangles and allow the robot to make 60-degree arcs. Of course, a triangular discretization complicates the data structures used to represent states, so we chose to stick with the grid representation.

We select a finite set of 90°circular arcs by allowing the user to specify the maximum speed for the robot's drive wheels, and the number of different evenly spaced wheel speeds to use to generate the action set. We then calculate a set of arcs that always drive at least one wheel at the maximum speed and the other at the closest speed to the speeds specified by the user that allows the robot to make a discrete state transitions. For an example action set of size 12 on a grid with cells of size 10 see Figure 3-5.

We can improve on the heuristic used in A* by taking into account constraints imposed by our limited action set. Gerry cannot move in a straight line from point

to point, but must instead move along grid lines or diagonally across grid lines by traversing a 90° circular arc. The shortest path that Gerry can possibly take bewteen two states, therefore, is either a straight line or a combination of a single straight line and a single 90° arc:

$$h = max(|dx|, |dy|) - min(|dx|, |dy|) + min(|dx|, |dy|) * \pi/2$$

Gerry must translate at least this distance. Furthermore, Gerry must rotate at least the minimum angle difference between the start and goal state. We can translate this angle into a distance by taking into account Gerry's wheel base length, b_{wheels}:

$$h = max(|dx|, |dy|) - min(|dx|, |dy|) + min(|dx|, |dy|) * \pi/2 + \theta_{min} * \pi/2$$

We call this heuristic the line-curve-pivot heuristic.

By searching for action sequences that correspond directly to actions that can be performed by the robot, we have made the problem of deriving a plan from the discovered path trivial to solve. The actions used to build the search tree are exactly the actions that make up the execution plan. Since every action in our action set is reversible, it is convenient to search backwards from the goal state to the initial state so that the plan actions can be read in order from the search tree by following backup pointers.

3.6 Searching from Both Directions

Unidirectional A* from the goal to the start is elegant and efficient but has several shortcomings. Unidirectional A* searches for the optimal path and only the optimal path. There may exist a path to goal at a much shorter depth in the search tree than the optimal one, and this path may be nearly the same length as the optimal path, but A* will ignore this path completely and continue searching until it finds the optimal solution. We would like a search algorithm that can take advantage of these short paths if an optimal solution is not found in a reasonable amount of time.

Another problem with a unidirectional search is that searches tend to fail most often because an obstacle is very near the start or goal state and is obstructing access to that state. There tends to be much more flexibility in moving around obstacles that are away from the start and goal states. If unidirectional A* starts at the goal state and the start state is blocked, the search will have to flood nearly the entire search space to determine that no path can be found. A failed search is very expensive and may last several seconds even for small search spaces.

Rather than use just a single A* search we instead perform a bidirectional heuristic search starting from the goal and the start states searching in opposite directions. Bidirectional heuristic search has the advantage of finding possible solutions very quickly. These solutions may not be optimal, but their error can be bounded. As a result, bidirectional search can often be stopped much earlier than standard A* once a solution that is close to being optimal has been found. Bidirectional search will also fail very quickly if either the start or goal states are blocked, because the corresponding search queue will become exhausted and the search will terminate.

For historical reasons, discussed fully by Kaindl and Kainz [3], bidirectional heuristic search has traditionally been labeled as having questionable performance, because the heuristics may push the search fronts past each other. A common analogy used is that bidirectional heuristic search is like trying to shoot a missle with another missle. This turns out to not be the case at all. While it is true that bidirectional heuristic search may theoretically expand nearly twice the number of nodes as A*, in practice, the bidirectional search fronts pass through each other and typically have their first encounter early in the search process. The main bulk of the computation time spent by bidirectional heuristic search is spent on satisfying the termination condition:

$$L_{min} \leq \max[\min_{x \in Open_{st}} f_{st}(x), \min_{x \in Open_{ts}} f_{ts}(x)]$$

where L_{min} is the cost of the best solution found so far, and $Open_{s't'}$ refers to the open list for the search from s' to t'. Intuitively, bidirectional heuristic search will terminate with the optimal solution if it has determined the best solution so far and there are no more possible better solutions on the search queues.

The termination condition for bidirectional heuristic search provides an upper and lower bound on the optimal path cost. We can therefore use a new termination condition:

$$L_{min} \cdot e \leq \max[\min_{x \in Open_{st}} f_{st}(x), \min_{x \in Open_{ts}} f_{ts}(x)]$$

where $0 \leq e \leq 1$. This new condition guarantees that the cost of the optimal solution will be no less than e times the cost of the best solution found.

We implemented and tested a variant of the BAA-add algorithm discussed in [3] where BAA stands for Bidirectional-A*-A* and the word "add" refers to a constant term added to the heuristic used. While BAA-add used the non-traditional technique of running one search until it filled its memory allowance and then switching directions once, we instead chose to use the more traditional cardinality criterion:

$$\textbf{if} \quad |Open_{st}| \leq |Open_{ts}| \quad \textbf{then} \quad Open_{st} \quad \textbf{else} \quad Open_{ts}$$

which selects a node to expand from the smallest queue. This helps to keep the search balanced.

The BAA algorithm is sped up using the Add method. The Add method makes use of the fact that the error made by heuristic for the search in the opposing direction can be used to increase the value of the heuristic function dynamically be a constant term. Let $d(t, b_i)$ for all nodes b_i on the fringe of the search from t to s be the error of the heuristic function from the fringe nodes b_i to t:

$$d(t, b_i) = g^*(t, b_i) - h(b_i, t).$$

The new heuristic function:

$$\begin{aligned} d_{min}(t) &= \min_i(d(t, b_i)) \\ H(a, t) &= h(a, t) + d_{min}(t) \end{aligned}$$

for all nodes a outside of the search frontier from t to s is also an admissible and consistent heuristic [3]. A similar heuristic $H(b, s)$ also exists in the opposite direction.

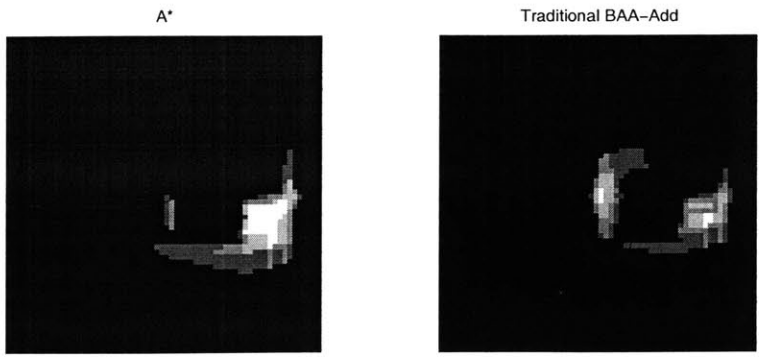

Figure 3-6: Nodes expanded by A* and tradition BAA-Add.

This new heuristic does not change the order in which nodes are removed from the queues; it only adds a constant term. However, it does help satisfy the stopping condition more quickly and provides a tighter lower bound on the optimal path cost.

A comparison of the traditional BAA-Add algorithm against A* shown in Table 3.2 confirms that BAA-Add spends an excessive amount of time validating that a solution is optimal. In some cases more than half of the search time is spent after the optimal solution has already been found! See Table 3.2. The disparity between A* and non-traditional BAA-Add was not as great in the experiments cited in [3]. This is probably due to the fact that the line-curve-pivot heuristic is almost perfectly accurate for several interations of the search algorithm, so d_{min} is zero and the Add method does not help as much.

Although traditional BAA-Add does not perform well when it is required to guarantee the optimal path, it does, however, find a solution path well before A* does, and quite often this solution is the optimal path anyway (as was the case shown in Table 3.2). We therefor chose to use traditional BAA-Add as our search algorithm with

Search Algorithm	Error Bound	Node Expansions	Path Cost
A*	NA	517	563.58
trad. BAA-Add	1	854	563.58
trad. BAA-Add	0	408	563.58

Table 3.2: Comparison of the number of nodes expanded by A* and traditional BAA-Add.

Data : s, t
Result: L_{min}
Create priority queues: q_s, q_t;
$L_{min} \leftarrow$ Fail ;
$\texttt{Cost}(L_{min}) \leftarrow \infty$;
while $\neg$*Empty*$(q_s) \wedge \neg$*Empty*(q_t) **do**
  if *Cost*$(L_{min}) \cdot e \leq \max[\min_{x \in Open_{st}} F_{st}(x), \min_{x \in Open_{ts}} F_{ts}(x)] \vee$
  Time() - *start* $\geq t_{max}$ **then**
    return L_{min}
  end
  $n \leftarrow \texttt{Pop}(q_s)$;
  if *Contains*(q_t, n) **then**
    $L_{min} \leftarrow \min(\texttt{L}(n), L_{min})$
  end
  if $\neg$*Contains*(Closed, n) **then**
    $q_s \leftarrow \texttt{Expand}(n)$
  end
  $n \leftarrow \texttt{Pop}(q_t)$;
  if *Contains*(q_s, n) **then**
    $L_{min} \leftarrow \min(\texttt{L}(n), L_{min})$
  end
  if $\neg$*Contains*(Closed, n) **then**
    $q_t \rightarrow \texttt{Expand}(n)$
  end
end

Algorithm 1: Traditional BAA-Add.

an additional time bound t_{max} after which the search is forced to quit. If no solution path has been found then a failure is returned, otherwise the best solution available is returned. The search may terminate before t_{max} if the error bounded termination condition is met. The pseudocode for our algorithm is shown in Algorithm 1.

In Figure 3-6 we show a comparison of our algorithm and A* by highlighting the states in configuration space that have been expanded. The configuration space is the same one that was build from the setup described in Section 3.1. The brightest pixels represent states that have been expanded for all four orientations. Dark pixels have not been touched. There are spaces between pixels because the search can jump states by performing a circular arc action. The two branches of traditional BAA-Add reach almost directly toward each other.

3.7 Directing the Search

Searching for optimal paths has the undesirable side effect of generating paths that scrape along obstacles on the way to the goal state. Slight errors in path execution could cause the robot to collide with the obstacle. As is discussed in the Plan Execution section, Gerry checks the path that he is currently following to see if it has become obstructed, and if it has, a new motion plan is requested. Noise in the sensor data can make the obstacle appear to jitter slightly such that a path which appears clear at one time step becomes obstructed on the next time step. As a result, the robot searches for new paths at an unnecessary rate.

To avoid generating paths that scrape along obstacles we can bias the search to look for wide paths, or paths that have a significant error margin. This is accomplished by adding a penalty to the cost of a given action that is inversely related to the distance to the closest obstacle:

$$f = length + p(w - o_{min})$$

where w is a positive integer specifying the maximum distance of the path of an action to its closest obstacle that need not be penalized and p is a constant specified by the

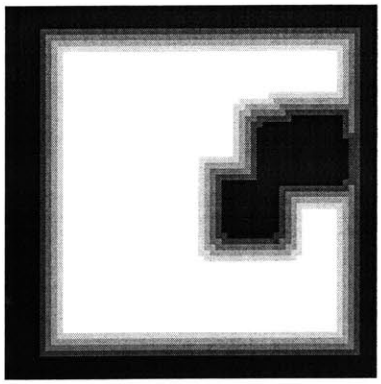

Figure 3-7: Visualization of distances to obstacles lookup table.

user.

We can calculate a look-up table that approximates the distance to obstacles by using a variant of the dilation algorithm used in graphics applications. First we create an array that is the same size as the configuration space and then initialize free cells to w and obstructed cells to 0. We then set all array elements in the neighborhoods of the 0 array elements to 1, then the elements in the neighborhoods of the 1 array elements to 2, etc until the procedure has been performed $w - 1$ times. Figure 3-7 visualizes this lookup table, where brighter pixels are further away from obstacles.

We can also bias the search against backwards actions. Backwards actions are generally undesirable because many robots, including Gerry, only have sensors that face forward and are blind to obstacles behind them. However, we don't want to completely disallow backwards actions, because they may be necessary to find a path to the goal, especially when the robot has nonholonomic constraints (for example, imagine that the robot is along a wall facing a corner). Biasing the search against backwards actions can be done by adding another penalty term, b, to the action length:

$$c_{backwards} = length + p(w - o_{min}) + b$$

where b is some number specified by the user. The advantage of adding penalty terms to the action cost is that a path will still be found even if a path with the desired properties does not exist.

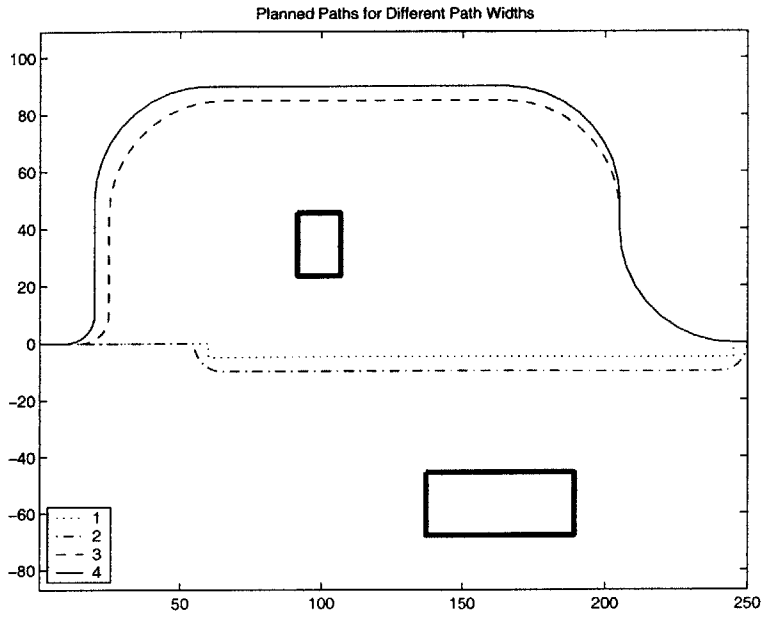

Figure 3-8: Paths found for different values of w.

The path-planning algorithm was tested in a simulation in a simple world with two box-shaped obstructions. The robot's initial state is $(0, 0, 0)$ and the goal state is $(250, 0, 0)$. For these tests, we discretized the configuration space using a 128 by 128 grid with a grid cell size of 5cm and 16 discrete evenly spaced orientations. For motion-planning we used a 64 by 64 grid with 10cm cells and 4 orientations. The robot was modeled as a rectangular bounding box of dimensions 78cm by 50cm by 56cm where the center of the bounding box was offset -14cm along the x-axis from the midpoint of the drive wheels and positioned even with the ground. The search algorithm used the set of 12 actions shown in Figure 3-5. The resultant paths for $1 \leq w \leq 4$ and $p = 50$ are shown in Figure 3-8.

3.8 Plan Execution

Plan execution is the process of taking a motion plan from the motion planning subsystem and then sending the correct control sequences to the robot so that it follows the desired path. The plan execution unit must execute the planned actions as accurately as possible and also deal with failure conditions.

Gerry's path planning subsystem generates plans that consist of a sequence of linear and angular velocities and the corresponding amount of time that the robot should be driven at those velocities. The plan execution system starts executing a plan by removing the first action from the plan, commanding the robot to drive at the linear and angular velocities specified, and then marking the completion time for that action. The plan execution unit then repeatedly checks if the action completion time has passed. If it has, the plan execution unit continues with the next action in the plan. Execution continues until the plan is empty.

Plan execution must be robust against the following sources of error:

1. Latency

   Since the plan execution unit can at best check if the action completion time has passed at some interval Δt, there is some latency between when the action was supposed to end and when the plan execution unit actually sends the next appropriate command. If this latency is too large, the robot overshoots on all actions and will drift from its intended path.

2. Obstructed Path

   Even if the motion-planning unit initially finds a valid path to the goal, this path may become obstructed due to an unobserved obstacle becoming visible, dynamics in the environment, or discretization error.

3. New Goal Location

   The user may at any time change the goal location. When this occurs the plan execution unit must abandon any currently executing plans, and request a new one to the new goal location.

4. Lost Goal State

   As the robot moves along a path to the goal, it may have to at first move away from the goal. This may cause the goal to go out of range. Since we consider all points out of range to be obstructed, the goal then may become perpetually obstructed.

By supplying the robot with some state estimation information, the robot can recover from execution error due to latency. State estimation data may directly originate from some sensor, such as odometers on the wheels or GPS, or it may come as a result of a state estimation algorithm such as Kalman filtering or particle filtering. State estimation allows the robot to compare where it currently is with where it should be. If there is a discrepancy, the plan execution unit can then attempt to perform a repair to the plan. When the plan execution unit detects that the robot has drifted too far from the expected path to still arrive at the goal location, it requests a new plan from the motion-planning subsystem.

For Gerry, we used the odometry sensors to estimate the robot's location. While odometry sensors are susceptible to multiple error sources, they are reasonably accurate over short distances. The plan execution unit also uses the odometry data to check the planned path to make sure that it is still free of obstacles. If the path has become obstructed, the plan execution unit requests a new path from the motion-planning unit.

3.9 Replanning

Since Gerry's path planning unit is designed to be fast and light weight, Gerry's default failure handing procedure is to request a new plan. Frequent re-planning can be problematic, however, as the robot may enter locally stable states that prevents it from approaching the goal. A simple example is as follows:

Consider the case where the robot is approaching a single hypothetical point obstacle straight on and the robot wishes to navigate to the other side to the star marker (see Figure 3-9. The robot is represented by the small square, and goal state is the grid cell marked by the diamond. For this example, the robot can only pivot or move forward. From its initial position there are two symmetrically equivalent optimal paths, which the robot arbitrarily picks from. As the robot executes its first action, which is to pivot, the goal location moves relative to the robot. Since the goal is further from the robot than the obstacle, it appears to move faster, and it

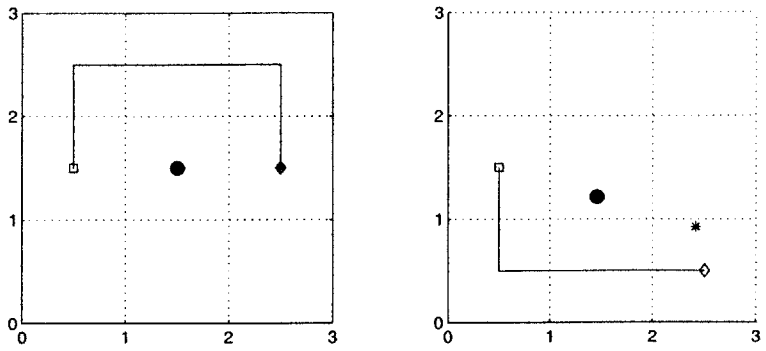

Figure 3-9: An example of situation with a single point obstacle that causes the robot to remain indefinately in a locally stable state.

transitions to a different state location before the obstacle does. As a result, there is a new shortest path that requires the robot to pivot in the opposite direction. The robot may stay at the same location pivoting back and forth indefinitely.

A couple of ways to fix this situation are to either stick with the original plan, or to compare the length of the current plan with the length of the new one. The new plan is only adopted if its length is the shorter than the current plan. This technique allows the plan execution unit to be opportunistic when a new, shorter path is encountered that could not be seen from the start state.

However, there are additional unstable states to consider. The data from the stereovision camera only gives a partial view of the world. The stereovision camera only provides information about the robot-facing boundary of obstacles that are in clear view. Other obstacles or parts of obstacles may be obstructed from view. As a result the robot can enter states where it oscillates without making any progress. The test setup described in Section 3.1 generates this behavior.

Over reasonable short periods of time, we can integrate the stereocamera data in an attempt build a more complete map of the world and thus avoid stability traps. The idea is similar to the time integration that was done to clean up the strereo data. The robot remembers a set of n processed occupancy grids and uses them along with the odometry data to reconstruct a world map. The solution was tested in simulation and worked well. Care must be taken, however, to avoid ghost images from moving objects in the environment.

Chapter 4

Topological Navigation

Here we describe a method for performing topological navigation using the localization and obstacle avoidance systems described previously. Although this method has not yet been implemented, we present it here as a possible direction for future work.

4.1 A View-based Approach

A topological navigation system represents the world as discrete states and transitions between those states. The hidden Markov model used for localization already provides a convenient discretization of space. However, it does not define the transitions, or in other words, a method to get between states. The obstacle avoidance system can navigate between (x, y, θ) states in the world, but the localization system as it stands does not store any spatial data with the locations.

To make transitions possible between locations, we adopt an approach similar in flavor to the view graph approach described in [2]. A view graph is a topological representation of a space consisting of local views and their spatial relations. Past view graph implementations have used single processed snapshots for the local views and a simple homing strategies to maneuver between views. In our system we have built a more abstract model for places and we will attempt to employ a similarly abstract model for performing transitions.

Intuitively, we want to build a set of models that we can use to calculate how

much a given image appears like it is on the path to a neighboring location. We wish to use sequences of images similar to those that were used to train the hidden Markov model for localization. Instead of labeling the images with the location that they were taken from, we propose to label them with the location that the robot is heading in the direction of. The training process is then virtually the same as it was for the localization system. For each location we train a model that calculates the probability that given the robot is in that location and given all observations since the robot entered that location the robot is heading towards a neighboring location. These new models, which we call the location transition models, can then be used determine if the robot is heading in the right direction to transition to a desired neighboring state.

We wish then to solve the problem of navigating between locations by driving the robot in the direction that looks most like it will lead to the desired location. The problem is analogous to trying to approach an invisible infrared beacon using a single infrared sensor. A single sensor reading does not by itself tell you in which direction to move, but multiple readings over time can be used to estimate a gradient that we can ascend.

We could attempt to ascend this gradient by driving forward only when the probability that the robot is facing the direction of the desired location is above a threshold specified by the user. If the current view does not sufficiently look like it will take the robot in the right direction, the robot can search for a better direction to travel in by spinning or wandering. Once the robot has found a probable direction, it again moves forward and the process repeats. Spinning to determine a good direction is an undesirable behavior that could be mediated or eliminated by using a pan tilt camera mount or an omnidirectional camera.

Searching for a sequence of locations to the goal location is a simple graph search problem. There remains the question as to what the arc weights should be. Location can vary in size and how easily they can be traversed, so a uniform arc weight over all arcs might not be the most desirable choice. It is proposed that this information could be learned from the training data.

While this is certainly not a complete treatment of the range of avenues to persue in building a topolocal navigation system based on vision, we hope it provides some "food for thought" and helps to lead to some exciting discoveries in this area.

Chapter 5

Conclusions

In this thesis we laid the ground work for a fully vision-based topological robot navigation system. We demonstrated the efficacy of vision-based localization using supervised HMM learning and global image features. Using stereocamera data we built a local motion-planning system using fast, robust search algorithms. In particular, we experimented with a novel use of bidirectional heuristic search to permit fast, semi-optimal searches. Finally we presented some ideas for combining spatial planning and topological localization into a complete system.

Bibliography

[1] Devin J. Balkcom and Matthew T. Mason. Time optimal trajectories for bounded velocity differential drive robots. In *IEEE International Conference on Robotics and Automation*, 2000.

[2] Matthias O. Franz, Bernhard Scholkopf, Hanspeter A. Mallot, and Heinrich H. Bulthoff. Learning view graphs for robot navigation. *Autonomous Robots*, pages 111–125, 1998.

[3] Hermann Kaindl and Gerhard Kainz. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research*, 7:283–317, 1997.

[4] L. E. Kavraki. Computation of configuration space obstacles using the fast fourier transform. *IEEE Transactions on Robotics and Automation*, 11(3):408–413, 1995.

[5] J. C. Latombe. A fast path planner for a car-like indoor mobile robot. In *Ninth National Conference on Artificial Intelligence*, pages 659–665, Anaheim, 1991. AAAI.

[6] Jean P. Laumond, editor. *Guidelines in Nonholonomic Motion Planning for Mobile Robots*. Springer, New York, 1998.

[7] Tomas Lozano-Perez. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, 2:108–120, feb 1983.

[8] Hanspeter A. Mallot and Sabine Gillner. View-based vs. place-based navigation: What is recognized in recognition-triggered responses? Technical report, Max Planck Institute for Biological Cybernetics, Tübingen, Germany, oct 1998.

[9] M. J. O'Neill. Evaluation of a conceptual model of architectural legibility. *Environment and Behavior*, 23:259–284, 1991.

[10] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[11] Eero. P. Simoncelli and William T. Freeman. The steerable pyramid: A flexible architecture for multi-scale derivative computation. In *2nd IEEE International Conference on Image Processing*, 1995.

[12] Antonio Torralba, Kevin P. Murphy, William T. Freeman, and Mark A. Rubin. Context-based vision system for place and object recognition. In *Internation Conference on Computer Vision*, 2003.