HYSTE: A HYbrid System for Thorough Exploration

Edward Gilkison Jones

April 30th, 2001

Abstract

This paper describes HYSTE, a system for thorough exploration of unknown environments by autonomous mobile robots. The system is based on a hybrid reactive/deliberative architecture that adds a middle layer for mapping and localization [11]. The reactive layer is implemented using a behavior-based motor schema approach. Motor schemata are parametrized and grouped into distinct behaviors, each of which causes slight but noticeable changes in navigation. A middle layer accumulates sensor information into an evidence grid and a topological connectivity graph. These maps are used by the deliberative layer to set goal points in unknown space to be achieved by the reactive layer. The deliberative layer can also select a behavior for the reactive layer, escalating the aggressiveness of the behavior selected if attainment of a goal point may require movement through dense obstacles or small gaps. Each layer is parametrized to allow for a maximum amount of durability and adaptiveness in real world environments. Results from several scenarios display the fully functional performance of HYSTE.

Contents

1 Introduction and Overview

Negotiating the real world is notoriously difficult for mobile robots. The real world changes quickly and unpredictably; it's also cluttered, difficult to sense accurately, and follows few discernable rules. All of these factors make programming robots for the real world a daunting task. A robot designed for operation in the real world must be endlessly adaptable, capable of dealing with many different situations, and able to act intelligently with incomplete information. These qualities are especially true of a mobile robot programmed for exploring and mapping an unknown area. Navigating in real world environments such as office buildings has been an open problem in robotics for years. This is a difficult problem because of the stringent requirements the real world demands of agents operating in it. This paper explores the issues associated with navigating and exploring unknown spaces with autonomous mobile robots.

The system described in this paper divides the task of exploring a completely unknown environment into two sub-tasks: selecting a place to explore, and physically exploring that place. This division forms the basis for the layers of the hybrid deliberative/reactive system [12, 2]. A low-level reactive layer based on a motor schema approach [2, 4, 3] directs the robot's motors to move through a space, attempting to achieve goal points set by the high-level, deliberative layer. A middle layer monitors the readings from the robot sensors, accumulating the sensory data into maps that can be used by the deliberative layer to select goal points in unknown areas. The reactive layer is based on a durable, reactive problem solving method inspired by animal behavioral observations. When encountering an obstruction interfering with the achievement of a goal point, the reactive layer will try a number of approaches to navigating around the obstruction. These multiple approaches mean that obstacles far more complicated than a simple garbage can or file cabinet can be autonomously negotiated by the reactive layer, allowing the robot to achieve goal points that lie beyond many complex obstacles. The deliberative layer processes the maps that the middle layer accumulates, applying a variety of techniques designed to intelligently pick a goal point for the reactive layer that will result in better exploration of the given region. The deliberative layer can also alter the behavioral strategy of the reactive layer, making the reactive layer more aggressive when it might result in the exploration of space that could not be otherwise explored. The middle layer not only accumulates maps of the obstacles and free space in the region, but also information regarding the connectivity of unoccupied regions, where the connectivity of two regions means that one region is directly accessible from another. Connectivity information can be used to navigate the robot safely and quickly through all known space.

The most important design paradigms of HYSTE are the adaptability and the autonomy of the separate layers. A number of parameters built into each layer can be altered as needed. This adaptability is most apparent in the reactive layer, which contains 50 parameters that can be altered by the deliberative layer to create different behavioral strategies for coping with all of the situations that an unknown environment may present. But these parameters and the goal point supplied to the reactive layer are the only interactions permitted between the reactive and the deliberative layers. Once both the behavior and the goal point have been specified, the deliberative layer does not attempt to micro-manage the reactive layer; instead, it lets the reactive layer alone to attack the problem it was designed to negotiate: the achievement of goal points and exploration of area along the way. The middle layer is similarly autonomous, interacting with the reactive layer only to obtain current sensor values, and with the deliberative layer to supply maps and compute paths through known space. The

deliberative layer can also adapt its own map-processing methods, to ensure that even in difficult situations a likely goal point can be found. Autonomy and adaptability can allow many layers of abstraction while letting each part of the system to function independently, permitting other layers to impact the way another layer functions, but never usurping the actual function. Figure 1 shows an overview of HYSTE.

The acronym HYSTE was chosen for two of its implications. The first is the homonymic suggestion of the word "heist." While exploring new area, the robot seems to be going normally about its business, and then suddenly begins to act very aggressively, acting, as it were,"up to no good." The robot appears to be guiltily fleeing the scene of some crime. The second implication stems from the observation that the letters HYSTE begin the word "hysteria," defined by Merriam-Webster's online dictionary as:

> 1 : a psychoneurosis marked by emotional excitability and disturbances of the psychic, sensory, vasomotor, and visceral functions
>
> 2 : behavior exhibiting overwhelming or unmanageable fear or emotional excess

Both definitions seem relevant. For the first, hysteria is defined as a psychoneurosis that causes perceptible changes in the actions of the affected individual. This seems akin to the changes that the deliberative layer effects in the reactive layer when it changes reactive behavior, particularly to an aggressive behavior. The robot operating with an aggressive behavior to get through a tight goal point certainly appears both excitable and disturbed. In terms of the second definition, the more aggressive behaviors brought to bear when the robot perceives that it is trapped appear much like the fight side of the "fight or flight" animal response to perceived enemies. These behaviors certainly seem to exhibit "overwhelming or unmanageable emotional excess."

A more thorough introduction to the reactive, deliberative, and middle layers directly follows. Next, background concepts are introduced that are necessary to thoroughly understand the approach utilized in the system. The following sections explicate the architecture of the system. First, the robot used in the research that created HYSTE is described, as well as the interface to the robot itself; the basic underlying architecture of the code is also described. Then the reactive layer is explained, focusing on the schemata governing reactive function and the ways they can be put together into cohesive behaviors. The deliberative layer is then considered at more length, discussing goal-point selection and evaluation, behavior selection, and adaptive ability. The middle layer will be referred to, but most of the work in developing it was done by Nathaniel Fairfield [11]. His thesis should be consulted for a more full explication of its capabilities. After the system is fully described, HYSTE's performance in several scenarios is illustrated and discussed. Finally, the conclusion will look at both successes and shortcomings of HYSTE, as well as future work planned on this system.

1.1 The Reactive Layer: An Introduction

When considering how to program an agent for navigation in an unknown environment, it seems natural to look at the methods of agents that have highly-developed and consistently successful solutions to this problem: animals [23, 2, 1]. Almost all animals depend on the ability to navigate in new or partially known areas for survival. Animals must navigate to forage for food, find shelter, and protect their territory. As navigation is so important to the survival of animals, natural selection

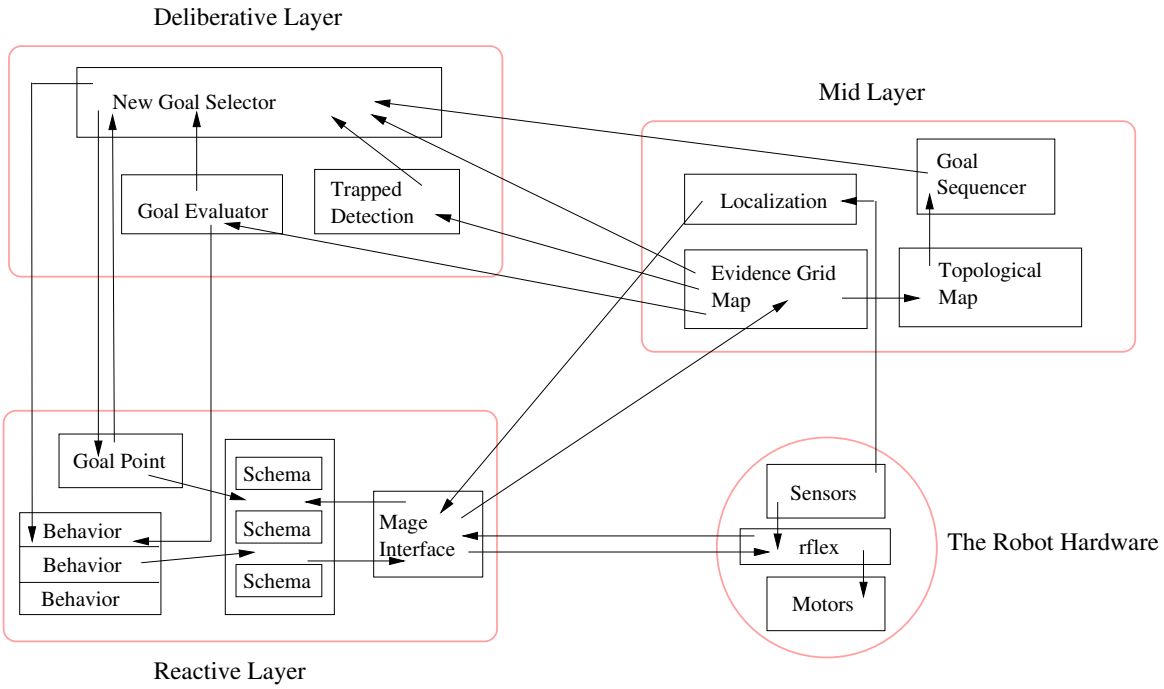

Figure 1: Underlying architecture of HYSTE.

has developed highly successful systems for real-world navigation. Consider an ant moving towards a known food source. It moves directly towards the source, but encounters an obstacle. The ant may try to climb over the obstacle, and if that doesn't work may attempt to go to the left of it, the right, or under it. The ant will try a variety of different approaches to circumnavigating the obstacle before it gives up on attaining the food source. It seems likely that the ant doesn't have a complex internal map of the area; it just knows where the food source is, and has an arsenal of different techniques for getting past any obstacle in its path.

The approach for low-level control used in this layer tries to emulate this ant-approach to navigation. The reactive navigation layer operates with no internal state, merely the knowledge of a goal and the ability to try a number of different approaches to get around an obstacle in its path. The implementation of the system is based on the kind of reactive problem-solving that gives the ant its impressive abilities to successfully navigate in the real world. If the reactive layer is not capable of getting to a goal point, it should mean that the goal point is physically inaccessible from all possible approaches near the robot. And even if the robot can't reach a goal point, it should explore a large portion of the area on the path to the goal, allowing a higher-level system to accumulate more information about the shape of the world.

The implementation of the reactive layer is based on a biologically-inspired behavior-based approach called motor schema [2, 3]. The motor schema approach works on the basis of pushes: an obstacle "pushes" the robot away, a goal point "pulls" the robot towards it. Other schemata can exert a force dependent on what each schema is trying to accomplish. The robot's world is composed of the forces at work upon it, which are summed together to create the behavior of the robot at any given moment.

The motor schemata used in HYSTE are designed to handle most situations that will face a robot moving towards a goal point. Simple obstacles are negotiated by the combination of the push towards a goal and away from an obstacle, which causes the robot to skirt the side of a compact obstacle. Longer obstacles, like walls, are negotiated using a local minima avoidance schema that turns the robot in one direction, generally, when encountering a wall head-on. Once the robot is turned, it moves forward, pushing constantly against the wall until it encounters an opening, a *de facto* wall follower. After a certain length of time, the behavior calling the schema will reverse the direction of the turn if another obstacle is encountered, insuring that both ways around the obstacle are explored.

These simple approaches can get the robot around all but the most complex obstacles. There will be times, however, that the reactive layer will not be able to get around an obstacle, even though it may be physically possible. Say, for instance, that a robot is in a room that is completely enclosed, except for a door, slightly ajar, that opens outwards. Most sensible navigation packages are designed so that the robot does not hit obstacles, though in this case, hitting the door would actually allow the robot to pass through to the outside. Thus the low-level reactive layer includes many adjustable parameters, which the deliberative layer can change when it deems that a new behavioral "strategy" will be useful in moving into an unexplored area. A behavior might, for instance, increase the magnitude of the push to a goal, which would allow the robot to bump into walls, or increase the push away from obstacles in an area that seem especially hazardous.

1.2 The Middle Layer: Overview

The middle layer (`mid`) is responsible for two very important aspects of HYSTE: mapping and localization. `Mid` was designed by Fairfield and adapted for use in this system [11]. For the deliberative layer to make intelligent planning decisions, it must have an accurate map, and an accurate estimation of the current position of the robot. `Mid` provides both of these.

Mapping in the middle layer is done using two methods. In the first method, an evidence grid implementation, originally designed by Moravec [20] and adapted for this project, keeps a likelihood of occupancy for every $n \times n$ area of the space, where n is about 5 cm. The occupancies are updated according to the most recent sonar data, and stored as a probability of occupancy, with one being certainly occupied, and zero being certainly unoccupied. `Mid` also constructs a more abstract topological landmark-based mapping system designed by Fairfield [11]. The landmarks in this system are of two types. The first type of landmark, called "breadcrumb" landmarks, are used to keep the robot localized. The robot is equipped with the ability to drop small, colored "breadcrumbs" periodically, recording the locations where a landmark was dropped. Whenever the robot sights a landmark, it consults internal odometry to get a best-guess estimate of which landmark has been sighted. Simultaneous sightings of multiple landmarks can localize the robot even more effectively, giving a position and rotation estimate that is generally quite accurate. The internal odometry of the robot is then updated using a Kalman-filtering algorithm [27] with the position of the landmark(s). Some localization strategy is necessary, as keeping an accurate estimate of the robot's current location is imperative for the formation of accurate maps, and Fairfield's methods offer a viable localization solution for any environment. The details of the localization process can be found in Fairfield's thesis [11].

The second type of landmark in Fairfield's system is called a connectivity landmark. Unlike localization landmarks, connectivity landmarks are not physical objects. The group of connectivity landmarks forms a graph of nodes and lines connecting the nodes. The connectivity landmarks are the nodes, with the lines between nodes demarcating the straight-line connectivity of the endpoints. Thus if two connectivity landmarks have a line connecting them, then, in the estimation of the system, the robot can travel directly between those nodes without encountering an obstacle. The connectivity graph is created as follows. The starting point of the robot is the first node in the graph. Any space the robot has actually occupied is judged to be unoccupied by another obstacle. So as the robot moves through the world, it leaves a trail of zero-probability occupancies behind it. A new connectivity node is created in the graph if a straight line to the current position from every existing connectivity node has to pass through non-zero probability space in the evidence grid. This means that it is impossible to reach the current position via a completely unobstructed straight-line path from an existing connectivity node. In this case, a new node is created in the last position that can reach both the current position and an existing connectivity node by a straight-line that does not pass through any non-zero-probability occupancy cell. This system guarantees that any cell in the evidence grid that has been visited by the robot either contains a connectivity landmark or can reach a connectivity landmark via a completely unobstructed straight-line path. It also guarantees that any connectivity landmark can be reached from another connectivity landmark by a series of unobstructed paths that lead from landmark to landmark, and that this sequence of paths can be discovered by simple graph-searching algorithms performed on the connectivity graph. Both of these features will be used by the deliberative layer as discussed below. Again, a more thorough description of connectivity landmarks can be found in Fairfield's thesis [11].

I added several features to these mapping schemes to make them usable in HYSTE. First, a simple array was added to `mid`, called the visitation map, which keeps track of where in the map the robot has actually visited. As this information is also used by `mid` to construct topological maps, it was not difficult to add a small map that includes time-stamped values in every cell that the robot has visited, and 0 in all of the others. Every time the robot enters a cell in the evidence grid, a function checks the cells entered against the visitation map. If that cell has already received a time-stamp, it is left alone. Otherwise, the current time is inserted as the time-stamp. This function lets the deliberative layer check how many new cells have been visited in a given time interval, as a way to determine the effectiveness of a behavior/goal-point combination. The details of the use of the visitation map are discussed below.

Finally, the topological mapping maintained by `mid` can be used to plan paths through known space. The nature of the connectivity mapping guarantees that both the current position of the robot and any position previously visited have a straight-line path through unobstructed space to a node of the connectivity graph. It also guarantees that all the nodes of the connectivity graph will be connected by unobstructed straight line paths. Thus the topological map should contain information that would allow a function, given two previously visited points, to create a sequence of goal points that will get the robot from some previously-visited point to any other previously-visited position along unobstructed paths.

I added a function `find_path` to `mid` that takes the current position and a previously visited target cell and creates a sequence of points that will guide the robot from the current position to the target position. To create the sequence, `find_path` first projects a line from the current position to the target position. If this line does not pass through obstructed space, then there is no need to consult the topological map, as there is already an unobstructed straight-line path to the target position. If this path is obstructed, then the topological map must be used. `Mid` already explicitly holds the index of the node on the topological graph that is connected to the current position. This node becomes the start node, the first point on the path to the target cell. This connected node information is not held for all previously visited cells. Thus `find_path` must project lines from the target cell to all the connectivity nodes until a line is found that does not pass through obstructed space. Unless `mid` malfunctions, a line with this characteristic should exist. When the line is found, the node it connects to becomes the target node, the second to last point in the path to the target cell. If this node is the same as the start node, then `find_path` returns a sequence of two points: the start node, and the target cell. If the nodes are not the same, then the topological map must be searched using a graph search algorithm. This function uses Dijkstra's shortest path algorithm [7] to find the shortest number of connected nodes that have to be traversed to get from the start node to the target node. As currently formulated, all edges are assigned a weight of one, though an actual metric distance could be assigned to each of the edges to give the shortest metric path instead of the fewest node path. Dijkstra's algorithm will return a sequence of nodes from the start node to the target node. `Find_path` then must merely add the target cell to the list to create the finished sequence.

1.3 The Deliberative Layer: An Introduction

The reactive, low-level layer is designed to move to goal points, exploring an area if the path to a goal point is obstructed, and trying numerous approaches to getting around obstacles. But to truly explore

a space, a reactive approach is not enough. A higher-level layer must decide where the reactive layer should move, and possibly change reactive strategies to insure that a space is maximally explored. Thus HYSTE also contains a deliberative layer, which evaluates the mapping information provided by `mid` and tries to find a goal point which will result in the exploration of unknown space.

Goal point selection in the deliberative layer is done using computer vision processing techniques on the most recent evidence grid data, provided by the middle layer. When the deliberative layer decides to try to find a new goal point, a message is passed to the middle layer to threshold the evidence grid data based on parameters provided by the deliberative layer. `Mid` then thresholds the evidence grid, putting the information into a PPM image file. After `mid` supplies this information, the deliberative layer has access to a color image that contains pixels corresponding to the cells of the evidence grid. The pixels, hereafter referred to as cells, are classified as one of three types: cells that are almost certainly occupied, cells that are almost certainly unoccupied, and everything else, the "unknown" cells. The classification of each cell is marked by a color, black for unoccupied, white for occupied, grey for unknown. These color classifications both serve as a ready way for functions to identify the classification of a certain cell, and make for clear image files viewable by the programmer. The thresholds used to determine which category a cell with a given probability belongs to are adjustable by the deliberative later.

The image then undergoes a shrinking algorithm [15] designed to replace isolated unknown pixels with unoccupied pixels, as they are likely to simply represent noise. The shrunken image is passed to a function used for frontier detection. This function examines the borders between cells judged unoccupied, and those judged unknown. A cell is designated a frontier cell if it is an unoccupied cell with an unknown neighboring cell and is more than a certain number of cells from an occupied cell. The cells designated frontier cells are then examined, and the largest group of connected cells is found by connected region extraction [15]. There may be hundreds of cells in a particular connected region, and not all of them are needed to establish a likely frontier. Thus if there are more cells than a certain maximum value, a small number of them are selected randomly as representative cells of that frontier region. The distance to the nearest occupied cell is computed for the representative cells using breadth-first search, and the single cell with the highest distance, the "hot" cell, is selected from that group. Breadth-first search along unoccupied cells with all unoccupied neighbors is then used to find the nearest cell to the hot cell that has actually been visited by the robot. Only cells with all unoccupied neighbors are candidates in the breadth-first search, as the robot cannot fit in an unoccupied area a single cell wide. If the breadth-first search from the hot cell cannot reach a cell visited by the robot, the hot cell and all the frontier cells in that connected region are erased as frontier points, and the whole process is repeated for the next largest connected region of frontier cells. If the number of pixels in the largest connected region is below a certain value, then the robot deems itself trapped, as not enough connected frontier pixels were found to constitute a likely frontier. Trapped behavior is discussed in more depth below.

If the breadth-first search does yield a visited cell connected by unoccupied area to the hot cell, then a new goal point is selected by projecting a line from that nearest-visited cell through the selected frontier cell, and beyond for a certain distance. The cell at the end of this projected line is selected as the new goal point. The deliberative layer uses the `mid` function `find_path` (described above) to create a sequence of goal points along unobstructed paths to guide the robot from the current

position to the nearest-visited cell. These goal points are sequentially passed to the reactive layer, and the deliberative layer sets the reactive behavior to `CRUISE`, a behavior designed to move through unobstructed space quickly. If a certain amount of time elapses and `CRUISE` has not achieved the next goal point in the sequence, the deliberator briefly attempts to reach the node using the `CRUISE_AGG` (for aggressive) behavior, a more aggressive version of `CRUISE`. If the goal point still cannot be achieved, the deliberator selects a new goal point. Generally, however, it expected that the robot can reach the nearest-visited cell. Once at the nearest-visited cell, the actual frontier goal recommendation is assigned as the next goal point. The reactive behavior `FAST_SAFE` is selected, a behavior is designed as a fast but safe explorer which will cause the robot achieve the goal point quickly if it is easily accessible, or explore obstructions that prevent the easy achievement of a goal point.

The ultimate goal of the deliberative layer is to explore new area. In accordance with this ultimate goal there are two standards of success for a given goal point/behavior pairing. Many goal points will be easily and quickly achieved by the `FAST_SAFE` behavior, which is one standard of success. In this case, a new goal point is selected by the methods above, and the process repeats. In some cases, however, a goal point may lie behind a serious obstruction. But a goal point/behavior pairing can still be successful in this case if new area is explored. To determine whether new area has been explored the deliberative layer uses a time-stamped map of the region being explored. Whenever a cell of the world is entered for the first time, a time stamp is assigned to that cell. The deliberative layer can thus determine a number of new cells visited in a certain time range by looking for all time stamps between the last time it looked at the time-stamped map and the current time. The deliberative layer will maintain a given goal point/`FAST_SAFE` pairing until the number of new cells visited falls below a threshold, suggesting that the utility of that pairing is at an end. At this point, the deliberative layer will re-evaluate the goal point. It maintains a record of the "hot" cell used to determine the current goal point. If there is no longer a connected region of several frontier cells in the region surrounding the old hot cell, a new goal point is selected. This lack of frontier cells near the goal point suggests that the old hot cell did not actually represent a frontier, and should not pursued any longer. As the selected goal point could not be reached easily, and the path leading up to that goal point is no longer a frontier, the goal point probably lies behind a wall and another frontier should be explored. If, however, a box around the hot cell still contains a sizable connected region of frontier cells, then the goal point is maintained. The continued presence of frontier cells could mean that the goal point is accessible, but achieving it may require a new, more meticulous or more aggressive behavior. If, for instance, the goal point lies directly through a gap that is too small for the reactive layer with `FAST_SAFE` parameters to get through, the gap should be marked with frontier cells. Thus if there are still a number of frontier cells the behavior is changed to `SLOW_SAFE` and the goal point maintained. If the reactive layer operating with `SLOW_SAFE` parameters cannot reach the goal point in a certain amount of time, and few new cells are being visited, the evaluation is repeated. If frontier cells are still found, the `AGG` behavior is set. If reactive layer with this behavior still can't reach a given goal point after a certain length of a time, a new goal point selected. Generally, the `VERY_AGG` behavior is reserved for trapped situations, in which actually attempting to move an obstacle out of the way might be useful.

The deliberative layer also possesses a level of parametrized adaptability for situations in which finding a likely frontier is difficult. The deliberator may evaluate the map by the methods above, and not find a sufficiently large connected region, suggesting that the robot may be trapped within a

confined space. In this case, the deliberator adjusts a number of parameters that govern the image processing used to select a goal point. Changing these parameters should yield a more "liberal" map, a map that marks what may potentially be very tight spaces as viable frontiers. These alterations may include raising the probability under which cells are classified unoccupied, as well as raising the probability necessary for a cell to be judged occupied. This will effectively shrink the occupied regions and enlarge the unoccupied regions, expanding any holes that may exist in the map. Next, the parameters governing shrinking can be altered to replace more unknown cells with unoccupied ones. By raising the number of unknown neighboring cells necessary for an unknown cell to remain unknown, areas of unknown cells that border on unoccupied cells will shrink as border unknown cells are replaced with unoccupied cells. Additionally, more unoccupied cells will be classified frontier cells if the distance a frontier cell can be from an occupied cell is decreased. Altering these parameters can create substantial connected regions in likely but tight places, frontiers that would not necessarily show up on a map processed with more conservative parameters but may be viable nevertheless. When the deliberator makes its parameters more liberal, it also changes the sequence of behaviors to begin with the `SLOW_SAFE` behavior, moving to the `AGG` behavior, and finally using the `VERY_AGG` behavior, which is parametrized to cause the reactive layer to attempt to push obstacles out of the way, creating its own frontiers. An exploring robot should be prepared to alter the environment to fully explore the space.

The next section presents background necessary for understanding the intricacies of the functioning of HYSTE.

2 Background

2.1 Potential Fields

The potential fields method as applied to robot navigation is an alternative to older, planning-intensive conceptions of navigation. Rather than directing motion with rules (i.e. "If the goal is to the right, then right motor gets 10 cm/sec right rotation"), or using a map to directly determine motor commands, potential fields navigation is a fast, continuous method of directing motion [18, 17, 2]. A potential field is composed of vectors. The motor commands of the robot at any position in a potential field correspond to the vector on which the robot is situated. Goals attract, and thus the goals will have vectors pointing towards them; obstacles repulse, and will be surrounded by vectors pointing away. Forces can be of constant magnitude on every vector in a system, or operate on a gradient. Many potential fields implementations have goal forces exert a constant force across the entire gradient, while obstacles will exert greater outward force the nearer the vector is to the obstacle. The vector at any given position is computed by summing all the forces exerted on that vector. In a simple domain with one goal and a single obstacle, the vectors will generally point directly towards the goal except in the area around the obstacle (Figure 2). A robot would move through this world towards the goal by following vectors primarily influenced by the goal position. If the robot encountered an obstacle, it would be pushed away from it by vectors primarily influenced by the repulsive obstacle forces, until it moved around the obstacle and could again move towards the goal.

There are two major difficulties with the potential fields method as applied to robot navigation. The first is that this method can be slow and require substantial internal state, as it appears that

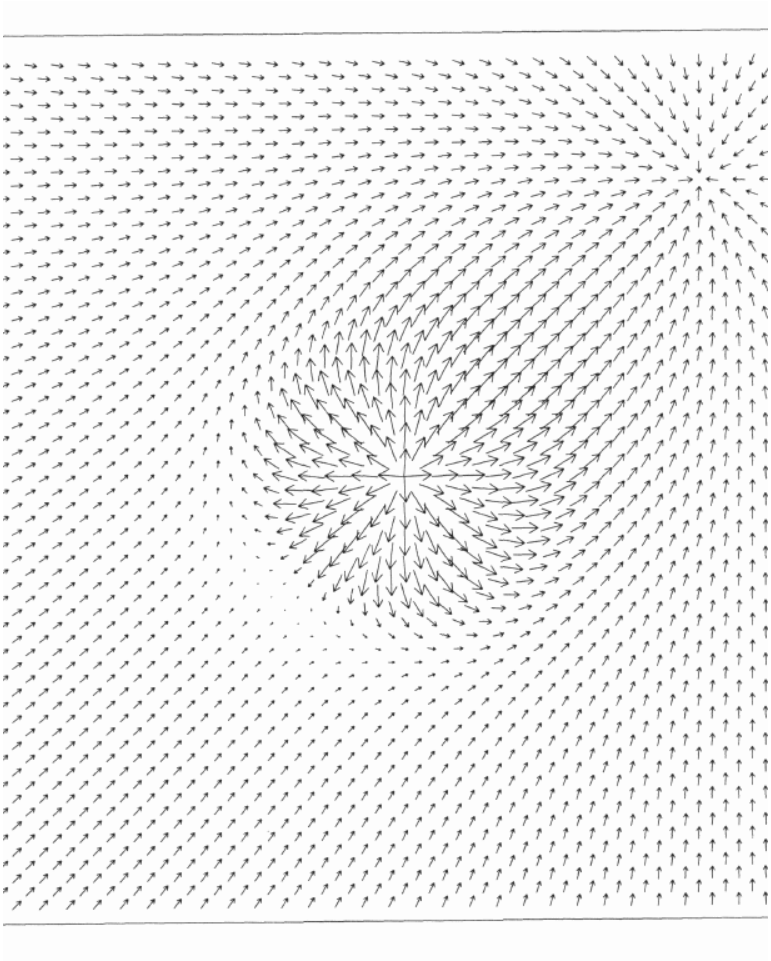

Figure 2: A potential fields representation of a domain with a single attractor in the upper right and a single repulser in the center [2].

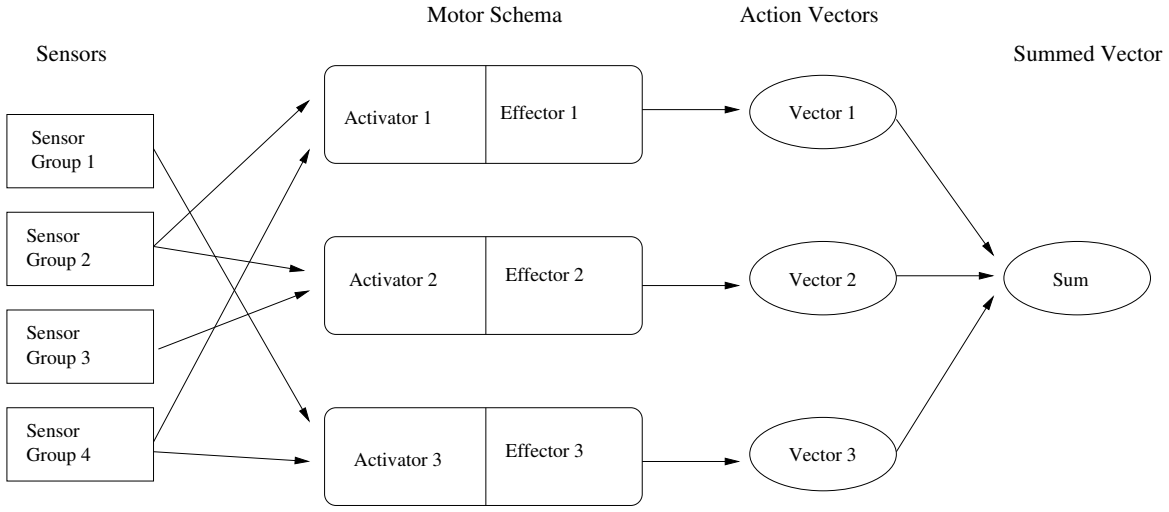

Figure 3: The motor schema approach. Information passes from the sensors into the activation portion of each schema, which examines a part of the sensor information and evaluates it. If the activation evaluation is positive, the effector part of the schema then computes a recommendation based on the sensors values and outputs a motor vector. All the motor, or action, vectors are summed and passed to the motors.

vectors must be computed based on a map of the entire region. But a potential fields method can be easily adapted to a reactive system. There is no real need to factor in forces outside of the robot's perceptual range, though true potential fields require computing vectors based on the entire field. A goal outside the perceptual range may need to be maintained, but obstacles that the robot cannot sense do not necessarily need to affect the motion vectors. Thus the vector accompanying the current position of the robot can be computed using only the sensory information from a single time step [4]. This vector computation using only current sensor readings makes this method reactive, in that it requires no internal state, and allows the vector on the current position of the robot to be computed quite quickly. In the reactive formulation of the potential fields method, no real potential field is computed, only the vector at the robot's actual position.

The second potential problem is local minima. The robot's motion vector at any moment is the sum of the forces affecting it. What happens, then, when these forces sum perfectly to 0? The robot could remain indefinitely in a single position, perfectly immobilized at a local minimum. If no force disrupts this equilibrium, movement may cease. Thus any potential fields method must employ strategies for coping with the problem of local minima. HYSTE has two such strategies, a time-varying noise strategy `fluct` and a turning strategy `stuck`. See Section 4.6 for a description of these strategies.

2.2 Motor Schema

The potential fields method is sufficient to create good behavior when the environment consists of only simple attractive and repulsive forces, but is not designed to support more complicated and

situationally-dependent forces. The motor schema method [2, 4] is an attempt to create a more general, behavior-based, conception of the potential fields method. A motor schema consists of an activation portion that examines the sensory data, and an effector portion that computes an action vector based on those stimuli (Figure 3). The effector portion computes an action vector based on a function that corresponds to the particular goal of the motor schema [2]. For instance, an avoid-obstacle schema might examine sonar data for especially low sonar readings, and recommend an action response that guides the robot away from obstacles. Each motor schema will examine a part of the sensor data and respond in an appropriate manner. Generally, a motor schema will have a continuous response, meaning that there can be an infinite number of action vectors for each motor schema. In the avoid-obstacle scheme above, an obstacle two meters away might result in a low magnitude vector pushing away from the obstacle, and a obstacle 30 centimeters away might result in a much stronger push, where the response is computed along a continuous range. A robot operating with a motor-schema based system may have many schemata, each of which examines a part of the sensory data and recommends an action vector. In this way sensory information from a variety of different modalities, such as vision, IR, sonar, and bump sensors, can be integrated into motor command decisions. A vision-based navigation system can examine camera data in its activation portion and recommend a motor vector based on perceived distance to the nearest obstacle, while the sonar data can also filter to motor commands in a similar fashion. The behavior of the robot is computed by summing all of the action vectors together and acting according to the result. By this method all the schemata are blended together to produce the actions of the robot at each time step.

If a single set of schemata does not seem sufficient to obtain the desired behavior, motor schemata can be clumped together into true behaviors, in which a behavior consists of a number of different schemata [2]. A wall-following behavior could consist of a goal-push schema that would push the robot beyond the wall, an obstacle-avoidance schema that would push the robot away from the wall, and a schema to turn the robot in one direction or the other when encountering a wall. In a true behavior-based system there would be many such behaviors, each consisting of a variety of schemata. A single behavior would be in effect at any one time, with a sequencer to evaluate sensor or state information to designate which behavior should control the robot.

2.3 Motor Schema Versus Subsumption

Comparison with the more widely know subsumption architecture can highlight some of the distinctive aspects of a motor schema architecture. The subsumptive style of behavior-based architecture was pioneered by Brooks [6] in attempt to find a fast, minimal state architecture for robot control. Traditional AI methods largely consisted of a sense-plan-act style of control, which was slow and often could not adapt to the rapidly changing and inhospitable real-world environments. Subsumption architectures consist of a behavioral hierarchy, with each behavior running simultaneously and in parallel. As in a motor schema approach, each behavior has independent access to the sensory data, and uses that data to formulate an action to be taken, or it may decide to recommend no action at all. But unlike the motor schema approach, a subsumption style architecture creates a hierarchy of behaviors, consisting of low-level behaviors that have no knowledge of the higher level behaviors. Coordination of behaviors occurs according to the priority hierarchy, in a winner-take-all approach. Thus if a more complex, high-level behavior with greater priority decides to act, it suppresses and subsumes the lower-level behavior, and the motor values at that time step will be those recommended by the highest-priority active layer. In the motor schema approach the action outputs of each of

the schemes are combined by summation, whereas in subsumption one behavior controls the motor outputs at each time step. In a subsumption architecture, the robot is doing one thing at once; in a motor schema approach the schemata are thoroughly blended [2].

Each type of architecture has advantages and disadvantages. Subsumptive design gives a single behavior control over the system, meaning that the behavior is given complete priority. This can mean that the agent does whatever behavior is currently in control very well, but it can also mean that the agent loses sight of the larger goals of the system. Motor schemata allow the blending of behaviors, allowing the behavior of the robotic agent to combine multiple goals. But a motor schema architecture must contend with local minima and situations in which having multiple goals means that none of them is fulfilled to the designer's satisfaction. Motor schema's advantages over the better known subsumption architecture are discussed in specific relation to HYSTE in the conclusion (Section 8.1).

2.4 Layered Architectures

Motor schemata can be used to create a successful reactive system, and can complete many tasks operating exclusively within a reactive framework, especially when combined with an able sequencer. Yet the motor schema architecture is completely reactive, which can ultimately limit its function in important ways. Reactive architectures are fast, cheap, and durable, but many complex tasks seem impossible to solve without planning and maintenance of an internal state, both of which are expressly non-reactive.

A so-called hybrid architecture attempts to combine planning and reactivity to harness the power of both methods [12, 2]. The most pertinent hybrid architectures for this paper are layered architectures that consist of a fast lower-level reactive layer and a slower, high-level deliberative layer, though HYSTE also adds a middle layer for map processing and localization. The deliberative layer examines an often extensive internal state, creates a plan, and passes that plan on to the reactive layer, which then autonomously executes it. Thus the layered architecture attempts to let the reactive layer execute well-defined tasks, as reactivity is well-suited to cope with a hostile and rapidly changing environment; the layered architecture also allows the deliberative capability to solve problems that seem to require planning and evaluation of an internal state.

Many different types of layered hybrid architectures have been implemented, but the basic concepts have remained constant. Sensory information passes from the low-level sensors and updates internal state kept by a higher layer. The sensory information is also used by the reactive layer to create the actual motor responses to the environment. The deliberative layer examines the accumulated sensory information in real time and can suggest a change in the overall goals of the system at any point. Many layered architectures will include a middle layer that is responsible for taking the goals established by the deliberative level and translating those goals into sequences of smaller actions that the reactive system then executes. Different systems will contain more or fewer layers and connect the layers in varying ways, but the basic schematic remains fairly constant over all hybrid architectures [12, 2].

2.5 Mapping

A reactive robot does not maintain an internal state, as the reactive paradigm demands that the response to the environment be based only on the sensory input of the current moment. The deliberative layer of a layered architecture, however, needs to accumulate data over time in order to make planning decisions. A primary method of organizing sensory data concerning the shape of a world is through a map of that world. Constructing a map requires that sensory data about the position of things in the world be collected over time and organized into a consistent representation. There are many types of maps that can be constructed using sensory information, as the sensory information can be accumulated using a number of different methods. Some maps attempt to represent the space in absolute metric terms, while others try to represent it using shapes, or even by creating graphs that represent spaces and the connections between them.

2.5.1 Spatial Occupancy Grids

A spatial occupancy grid is an example of a model that attempts to represent the world in absolute terms without trying to identify any individual objects. A spatial occupancy grid consists of a two-dimensional grid of cells, each cell corresponding to a small region of the world. Each cell contains an occupancy value, a value that represents whether or not the cell is occupied by an object. In the spatial occupancy representation of most interest in this paper, an evidence grid (originally designed by Moravec [20]), the cells each contain a probability that the cell is occupied. A cell that is almost certainly occupied will contain a high value and one that is almost certainly empty will contain a low value. As the robot moves through the world, the sensory data that it accumulates is used to update the probabilities of the cells. Evidence grids attempt to compensate for the noise and random errors that often accompany sensors' interactions with the real world. A single free reading will not cause a cell to be judged forever empty. Only repeated, consistent readings will cause a cell's occupancy probability to change substantially, which ensures that mistaken readings and noise will not compromise the map representation. Additionally, occupancy probabilities tend to decay, slowly returning to an unknown status unless their probabilities are frequently updated, mirroring the real-world notion that information becomes stale unless frequently reconfirmed. Spatial occupancy grids are useful because they do not depend on the identification of any particular objects or features of the world, and take a pragmatic attitude toward the reliability of actual robot sensors. However, there are drawbacks to this kind of model. It requires substantial processing power, requires a good localization scheme (Section 2.6) for real accuracy, and is difficult use for planning given the considerable amount of uninterpreted data in the model.

2.5.2 Topological Representations

Spatial occupancy grids attempt to define the world in absolute terms, such that a grid cell or a set amount of distance corresponds to a metric measurement. Unfortunately, sensor noise and localization issues often skew the metric data that the grids need to maintain accurate representations of a space. In addition, spatial occupancy grids are such a low-level way to represent the world that relying exclusively on them for all deliberation can be quite time consuming. Topological mapping seeks to avoid the pitfalls of metric representation by focusing instead on regions or objects and the connections between them; it abstracts important information from the world into a compact representation. For robot navigation, the knowledge that one place is accessible from another may be more important than precise metric knowledge of an environment. Thus, the representations of a topological map

depend largely upon the notion of connectedness of regions. A topological map may look like a graph, with vertices representing a feature of the environment, such as a landmark (discussed below) and edges representing connections between those landmarks. An aspect of metricality will often be added to a topological representation, so that vertices will be situated relationally in a space and edges will have length and orientation proportional to their metric equivalents [8].

Topological representations are most useful when knowledge of the connectivity of features is a primary goal of mapping the environment. This connectivity information can be held quite compactly by a topological representation, and can be searched easily using all the standard graph search techniques. The main difficulties that a topological representation faces involve the creation of the representation. Creating schemes for feature detection and identifying connectivity in non-ideal environments can be extremely difficult. An ideal landmark for a topological scheme is one that is static, uniquely identifiable upon repeated visits, and can be used to deduce position and orientation. A final requirement is that these landmarks be easy to sense by the robot, but with few false positive identifications. These stringent requirements combine to make landmark detection no easy task. The most complete systems use vision and other robotic sensor systems to identify landmarks, but these systems are subject to all the normal difficulties facing robotic sensing systems [21]. These systems also face the added difficulty of dealing with sensorially homogenous environments like office buildings, which hamper unique feature identification and can lead to many false positive feature recognitions. Another method of landmark creation attempts to overcome the difficulties associated with sensing obstacles in homogenous environments by physically placing uniquely recognizable objects, thus creating landmarks that resemble the ideal ones above [9]. Finally, an easier method to engineer for landmark identification is the doctoring of an environment with obstacles that are readily sensed and identified, like bright patches of color or bar codes [25].

2.6 Localization

All mapping methods must consider how the positional information of the robot is maintained [14]. The simplest method of localization on most robots is the use of dead-reckoning. Dead-reckoning uses the wheel encoders of the robot, which record the number of number of forward and backwards revolutions each wheel has made. This information is used to compute the position of the robot relative to the starting position. This method does not reference the outside world in any way: hence the name dead-reckoning. Unfortunately, the wheel encoders on any robot are prone to error, and these errors are compounded over time if not corrected. Thus, the robot's estimations of its own position monotonically worsens as time passes in a system that uses only dead-reckoning. A robot that cannot reference the outside world to correct its estimation of its own position will inevitably become so inaccurate in its positional estimation that maps constructed using the faulty estimation will be virtually useless.

Because the accurate estimation of position is so important for mapping, every robotic mapping system that requires accuracy in the long term must use some method of localization, using information from the outside world to make position estimates more accurate. Ideally, some system like Global Positioning could be used to determine the position of a robot to a very high degree of absolute accuracy, but unfortunately such systems do not function indoors. Many methods of localization have been implemented, but the ones of most interest for this paper involve the use of landmarks to help

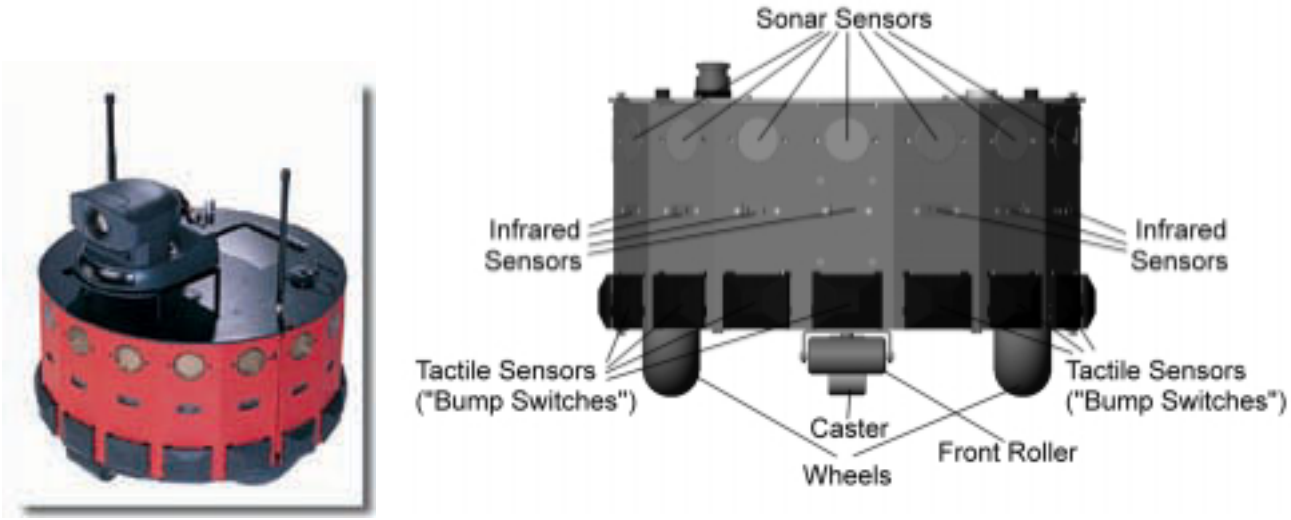

Figure 4: Left: The Magellan Pro. Right: A schematic excerpted from the Magellan Pro documentation provided by RWI

determine position more accurately [19, 5]. If consistent, static, easily detectable landmarks can be discovered or placed close to the starting position of the robot, then their positions can be recorded while dead-reckoning errors have not had time to compound. Subsequent visits to these landmarks can then be used to localize the robot. If vision is being used to determine landmarks, then depth and orientation information can sometimes be deduced from obstacles that are a substantial distance from the robot. If two or more landmarks are discovered simultaneously, then triangulation methods can be used to determine the position of the robot, if not the orientation [19].

3 System Basics

This section describes the hardware and software framework on which HYSTE rests. The first section describes the setup of the Magellan Pro and II, two robot models from RWI. All the research presented in this paper was performed on these robots. Next follows a brief description of the `Mage` API written by Fairfield, which is used to actually communicate with the robot. Functions in the `Mage` API return sensor readings from the robot sensors and communicate the motor commands determined by HYSTE to the actual robot hardware interface. A brief description of the thread-based, modularized system REAPER built by the Swarthmore College 2000 Robotics Team follows. REAPER provides the paradigm that organizes the system's modules and specifies the methods of inter-module communication. Finally, the modules themselves are considered, with a description of the contents of each module and how they interact.

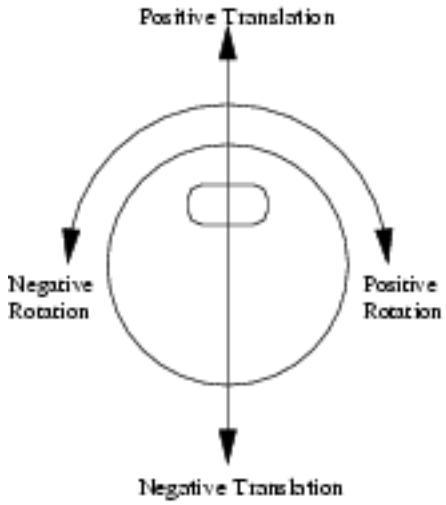

Figure 5: The effects of positive and negative translation and rotation motor commands on the positional configuration of the robot. The square with smoothed edges at the front of the circle (representing the camera) marks the front of the robot.

3.1 The Magellan: Hardware and Interface

The Magellan Pro (Figure 4) and the Magellan II are virtually identical robots designed by RWI. The Magellan Pro is slightly older, with a Pentium-II chip and a Sony Pan-tilt Zoom color camera. The newer Magellan II has a P3 chip and a Canon PTZ camera. Each robot has identical sensors, with 16 sets of tactile (bump), IR, and sonar sensors arranged in columns in an equidistant ring around the robot. The robots have two motors each, with differential drive, so that the robots can turn almost completely in place. They have a top speed of 1 m/s, though for safety's sake they are never really allowed to go that fast.

Each of the robots is running RedHat Linux on the onboard processor, which makes the programming environment like any other Linux machine. All the code for the system is written in C, as it is fast and low-level, two things that are required for robot programming. RWI offers an interface `Mobility`, but it was found lacking in important areas. For instance, `Mobility` did not implement access to the IR sensors. Additionally, the most thorough approach to programming robots requires having complete control over the most basic functions of the robot and building up from there. Thus the `Mage` communication interface gives virtually direct access to the robotic hardware. There is a hardware controller on the robot called `rFlex`, that actually interfaces directly with the robot hardware, but for all practical purposes `Mage` is the lowest-level interface possible on the Magellans.

The `Mage` interface to the robot takes care of several important functions. First, a function allows HYSTE to directly specify a motor command for each motor, giving total control over the motors. A function in `Mage` accepts two arguments, a translation value and a rotation value. This function directly sets the rotation and translation velocities and accelerations of the motors according to the

arguments the function was called with. The arguments should be in mm/sec, such that a positive translation moves the robot forward, a negative translation moves it back, a positive rotation turns it toward the left, and a negative rotation turns it toward the right (Figure 5). Another useful and necessary function in `Mage` queries the sensors and fills a `State` array with all the current values of the sensors. This array is placed into the reactive layer's shared memory segment so it can be accessed by the middle layer for map construction. Finally, this `State` array also contains the most current information from the wheel encoders, which count the number of revolutions made by the wheels to determine the current relative position of the robot. All of these capacities, essentially input to and output from the robot hardware, are handled by the `Mage` interface.

3.2 The Software: Shared Memory and Modularity

The design of the software is based on the REAPER architecture created last summer by the Swarthmore Robotics Team [22]. The system is implemented in a number of modules, discussed below. Each of these modules contains a main process, although the modules are never intended to run independently. The modules all run concurrently by using `pthreads`. A single module spawns the threads for all of the other modules and creates a large common shared memory structure. All of the spawned modules start up independently until all the modules are running concurrently and independently. The modules communicate via the shared memory structure, to which all modules attach immediately after they have been spawned. Each of the modules has a structure in the shared memory dedicated to that module, although occasionally two modules may share a single structure. Each of these module structures contains some values that are designated input and others that serve as output. The input variables can be altered by other modules to communicate with the module, and the output variables are filled by the module and can be read by other modules. The robot system at any time is executing many tasks concurrently. Having all of these tasks execute sequentially as part of a single process seems to violate the immediate response necessary for a robot to move through the world. Currently, six modules are running; clearly, these modules do not directly correspond with the three layers of the system.

3.2.1 `control` and `navigation`

The reactive layer consists of two modules: `control` and `navigation`. `Navigation` contains all of the sensor processing and obstacle avoidance capabilities of the system, and interfaces directly with `Mage`. It also interacts heavily with `control`. `Control` was arbitrarily chosen as the module that starts all the other modules. Additionally, it contains all of the behavioral definitions used by the deliberative layer, as well as the schemata functions. The schemata recommendations are totalled, and passed via shared memory to `navigation`, which sums them with the obstacle avoidance recommendations and passes the values to `Mage`. `Control` also checks the current position of the robot, setting a flag parameter in the deliberative shared memory if the goal point has been achieved. `Control`'s final capacity concerns behaviors. When the deliberative layer changes the behavior, `control` notes the change and fills in all the new parameter values in the reactive shared memory portion. Finally, `control` is responsible for varying the direction that the turning local minima avoidance schema `stuck` turns, as discussed in more depth in Section 5.2.

3.2.2 `vision`

The vision system of the robot has its own process and shared memory structure. The capabilities of the vision module are considerable, and it will be far more heavily used when this system is adapted for an actual application. For now, its sole use is as a landmark detector. Any module can set a vision task, such as face detection or motion detection. Vision will execute these tasks, and place information regarding the results in the shared memory structure. In the current implementation of HYSTE the only module that uses vision is `mid`.

3.2.3 `mid`

`Mid` contains all the mapping functions in the system, and was primarily designed by Fairfield [11]. All of the maps are held in local memory, though `mid` does supply the `plan` module with mapping information upon request, putting an image in the deliberative shared memory segment and supplying the most current map of the places visited by the robot. The entire middle layer is contained in `mid`.

3.2.4 `plan`

As the name suggests, the entire deliberative layer is contained in `plan`. Its interactions with the other modules are as noted.

3.2.5 `monitor`

As REAPER grew more and more complicated in the summer of 2000, the shared memory grew with it [22]. It became important for debugging purposes to be able to examine the contents of the shared memory to determine whether or not the robot was functioning correctly. Thus `monitor` was created using Motif. The monitor module constantly polls the shared memory structure, updating the monitor function on the screen with the information. Thus the shared memory can be viewed at any time.

4 The Reactive Layer

The next section contains information regarding the schemata that govern the function of the reactive module and the groups of parametrized schemata known as behaviors. There are over 50 parameters that can be altered to create different configurations for the five schemata, and much research effort was spent in slightly altering schemata parameters and viewing the results. Effort has been made to document these observations for future users of HYSTE, and more extensive material concerning both schemata functions and behaviors resides in two appendices, Appendices A and B. An abbreviated discussion of all aspects of the reactive layer follows, including results of goal point achievement experiments conducted using just the reactive layer.

4.1 Sensor Processing

The most basic segment of the reactive layer is the sensor processing that occurs within the navigation module. `Navigation` is the primary communicator with the `Mage` interface and processes most of the sensory data used by the rest of the reactive system. A utility queries the `State` vector filled directly by the `Mage` interface. These variables contain the most current data from the sonar, IR, and tactile

sensors. The data from the bump sensors is placed directly into the reactive shared memory structure to be examined by the `bump` schema as discussed below. Each of the IR and sonar sensors returns a value corresponding to the distance of the closest object in that sensor's range. A low reading indicates that the distance to the nearest object is small. For the 16 pairs of IR and sonar sensors, the minimum reading from the two is computed, taking into account a threshold for IR accuracy. IRs tend to be more accurate at closer ranges, but lose their accuracy at greater distances. Sonars lose their accuracy at close distances, but have a much longer range than the IRs. Thus if the IR reading is lower than a certain parameter `ir_thresh`, it is reported as the lowest reading for the pair, while otherwise the sonar reading is assumed to be the more accurate of the two.

4.2 The Obstacle Avoidance Schema

Obstacle avoidance is integrated into the navigation module, as it is difficult to imagine a scenario in which some flavor of obstacle avoidance would not be desirable, although it effectively acts as a normal schema, making recommendations based on sensor values. But the obstacle avoidance schema could be parametrized in a such a way as to effectively turn it off, or even have obstacles exert an attractive force. Obstacle avoidance is implemented in a reactive potential fields approach as described in Section 2.1. Obstacles in front of the robot push the robot backwards, objects to the left push the robot right; left objects exert a rightward force, and objects behind the robot push it forwards. As the sensors are arranged equidistantly around the robot it becomes a bit unclear what constitutes the front of the robot. For the purposes of this paper, front is defined as the direction the camera is pointed in its initial configuration, which also lines up with the wheels. This means that there is a single sensor pair pointed directly at the front, two other sensor pairs pointed at the front at a slight angle, two more sensor pairs at a slightly more extreme angle, and so on. As the goal of obstacle avoidance is not to hit anything, it is advantageous to include as many sensors as possible in the calculation of movement vectors, but not so many that the robot's progress through tight spaces is limited. Thus the current schema involves a variable weighting of the sensors for each direction, such that the most important sonars receive the highest weightings in the vector recommendation computation in each direction.

A number of parameters affect the function of the obstacle avoidance schema, and the exact function of each is discussed at considerable length in Appendix A. One group of parameters, `care` values, specify the distances at which obstacles begin to be avoided. Another group called `sensor_model` sets sensor weightings that control how important obstacles perceived by a sensor in a particular direction are to the final avoidance recommendation. These weightings combine with the sensor reading to create a scaled value that is multiplied by a third group of parameters, `max` values, that translate the scaled value into units which make sense for the motors. The final translation and rotation avoidance recommendations are summed with all the other schemata to produce the motor commands for a given time period.

4.3 Schemata Overview

As discussed in the introduction and background sections, motor schemata monitor incoming sensory information and produce movement vectors based on that sensory information. In this implementation, each motor schema gives two recommendations, a translation recommendation and a rotation recommendation, although some schemata only ever have a non-zero recommendation for one of the two. Behaviors (Section 5) can turn schemata on or off as they choose by setting an integer in

a schema_on array to one. Every time step the schema_on array is processed and the appropriate schema function is called if the corresponding integer of schema_on is set to one. Each called schema function examines part of the sensory data and makes translational and rotation recommendations, which are then collected in an array. Finally, the recommendations are summed and then placed in a store which can be read by the navigation module. During the operation of navigation it sums the total output of the schemata with the recommendations of the navigation-situated object avoidance schemata. All of the schemata outputs summed together form the motor commands that the robot will execute in that time cycle.

Every effort has been made to fully generalize and parametrize these schemata to keep them as broadly useful as possible. By altering parameters associated with a schema, the function of the schema can change pronouncedly, allowing behaviors to tailor the schemata on a highly individualized basis. Thus most applications should be able to use these schemata for any task that requires a goal schema or a bump schema or any of the other schema. The schemata used in HYSTE will be parametrized in particular ways to create behaviors useful for autonomous exploration, but many other behaviors can be formed from differently parametrized schemata.

The following sections describe the inner workings of the schemata: how they examine sensor data to produce action vectors, and the parameters that can be altered to adapt the function of the schemata to many different behaviors. More information on each schema's parameters can be found in the schema Appendix A.

4.4 Bump Schema

The first and perhaps simplest schema is the bump schema, which makes a recommendation only if one of the bump sensors is depressed. When a given bump sensor is depressed, the bump schema multiplies a parameter bump_force by the sensor weightings for that sensor grouping (also consisting of an IR and a sonar sensor as shown in Figure 4). This creates both a translation and a rotation recommendation that will move the robot away from the bumped obstacle for several cycles. Behaviors may need to react to the depression of a bump sensor in different ways, and the parameters for the bump schema can actually be modified to push into obstacles when bumped (see Section A.3).

4.5 Goal Schema

The goal schema is perhaps the most important schema given that the ultimate objective of this schema system is attaining goal points. The goal schema uses the current (x, y, θ) position of the robot and the (x, y) position of the goal to compute a goal heading h using the atan2 function in the math.h library. This heading consists of the angle from the current heading to the goal. A heading of zero means that the robot is pointed directly at the goal. This value can be used to determine a rotation that will eventually cause the robot to point directly towards the goal if no other forces are working upon the robot. Translation recommendations are kept at a constant positive value, as the robot should move forwards unless actively prevented from doing so.

Once a heading has been computed for the robot, the goal schema attempts to make recommendations that will return the robot to the proper heading, zero. A number of parameters can alter the

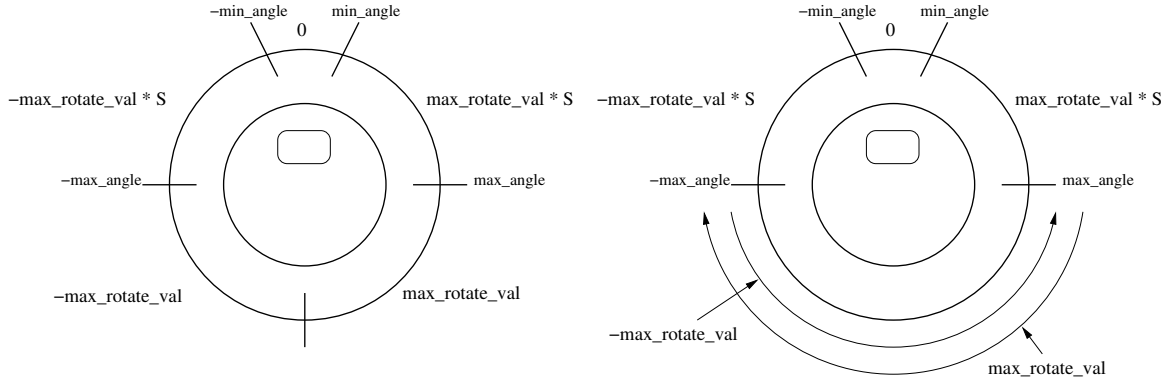

Figure 6: Left: Normal `goal` schema recommendations when the goal is at a given angle in relation to the robot. S is a value $(a - \texttt{min\_angle})/(\texttt{max\_angle} - \texttt{min\_angle})$, such that a is the angle to the goal. Right: `Goal` schema recommendations when `goal_follow` is set to 1.

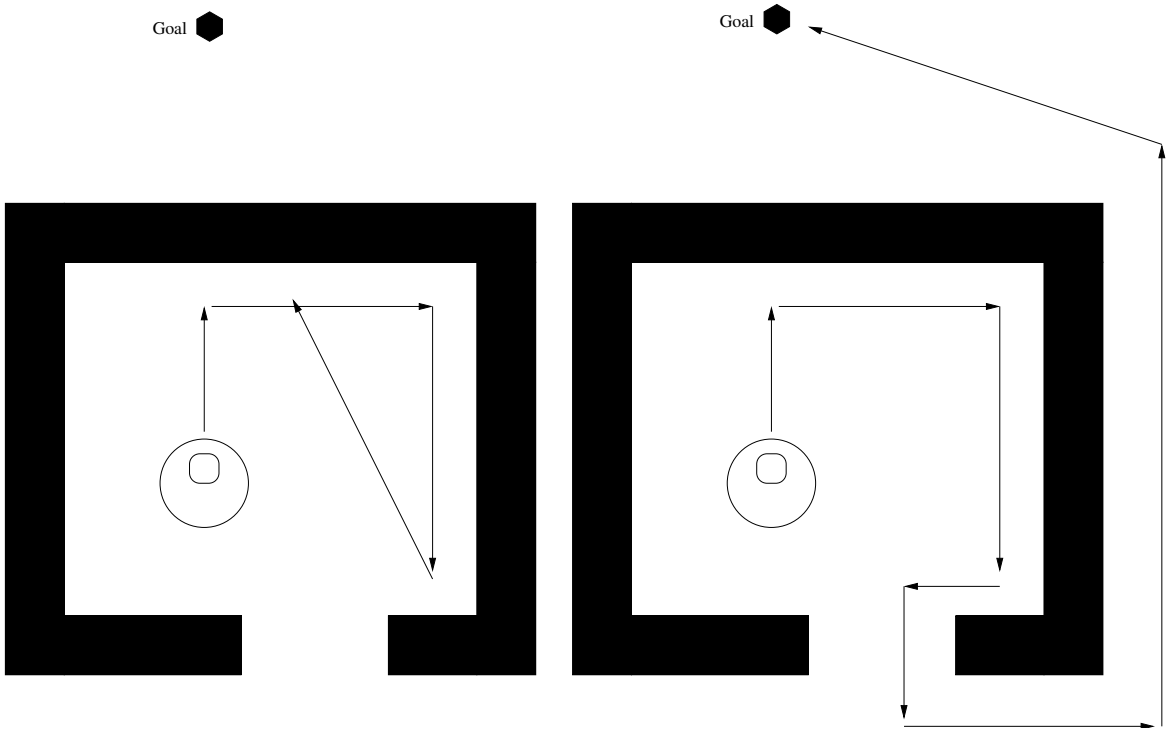

Figure 7: Left: The `goal` schema with `goal_follow` set to 0. Right: The `goal` schema with `goal_follow` set to 1.

nature and magnitude of this recommendation. The `goal` schema scales the rotation recommendation in accordance with the angle from the current heading to correct heading (Figure 6). Thus if the robot is heading in a direction that is 175° away from the goal, the goal rotation recommendation will be much larger than if the robot is moving in a heading only 10° from the correct heading. A parameter `max_angle` determines at what angle a maximum rotation will be recommended. Any heading greater than this maximum angle and less than 180° will result in a maximum push in a particular direction. Another parameter `min_angle` allows a bit of leniency around the correct heading. If the current heading is less than `min_angle` from the correct heading, a zero rotation is recommended. When a scaled value has been determined using the `min_angle` and `max_angle` values with h, the result is multiplied by another parameter `max_rotate_val`, which translates this value into a proper motor command. Finally, the correct direction for the recommendation is added. More details about the computation and these parameters can be found in Section A.4.

Additionally, the ability to simulate wall-following behavior has been implemented in the `goal` schema. Suppose that the robot is in the bottom of an inverted standard U-shaped curve, with a goal point directly beyond the cup of the U. It moves toward the goal until it gets stuck in a local minimum created by the cup of the U, and the stuck scheme turns it to the right. Facing the right side of the U, it gets stuck again, turns to the right, and begins following the side of the U. If at any point the robot's orientation to the goal gets more than 180 degrees the `goal` schema will promptly reverse the direction of the push, causing the robot to whip around to the right, and head straight back into the center of the cup. This scenario constitutes an infinite loop that can be difficult to detect and escape (Figure 7 (left) and Figure 10 (right)). In this case, a better behavior for achieving the goal would continue to push against the right wall until the end of the U is found (Figure 7 (right)). This may mean, however, continuing the leftward push against the wall even if it leads directly away from the goal. A robot with this type `goal` schema can achieve goals that could not be achieved with the normal goal schema (Figure 7 (right) and Figure 10 (left)).

The `goal` schema in HYSTE has a mode that causes it to function as a wall-following goal schema. If `goal_follow` is set to one, then the `goal` schema will recommend rotations that will cause the robot to follow walls, as in Figure 7 (right). In contrast, a `goal_follow` value of zero will cause the goal recommendation to switch sign at 180°, creating the non-following behavior shown in Figure 7(left). If the `goal_follow` variable is set to one, a goal angle value above `max_angle` will cause the `goal_following` variable to be set to one. When this happens, the `goal_max_rotate` with the appropriate sign is assigned to `goal_follow_push`. The goal schema will continue to recommend this value until one of two conditions occurs. If the `goal_follow` value is set to zero by a behavior, then the `goal` schema will revert to non-follow performance. Otherwise, any time the absolute value of the angle to the goal drops below `max_angle`, `goal_following` will be set to zero and the rotation recommendation will be computed as usual. Thus following simply extends the maximum rotation push in a certain direction from (`max_angle` to 180 degrees) to (`max_angle` to (360° − `max_angle`)) (Figure 6). This `goal` schema mode expands the situations in which the reactive layer functioning alone can attain a goal point that would otherwise be unattainable.

4.6 Local Minima Avoidance: `Stuck` and `Fluct`

One of the most formidable problems associated with potential fields type navigation are local minima: What does the robot do when all the forces impacting it sum to zero? There are scenarios in which both

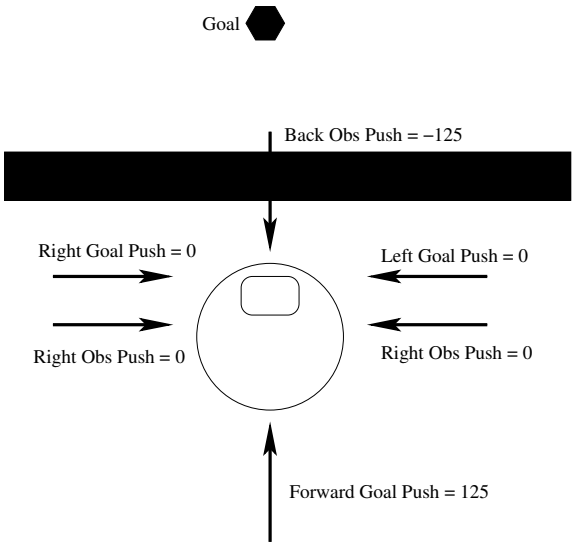

Figure 8: A local minimum, where all the `goal` and obstacle avoidance recommendations sum to zero.

the translation and rotation vectors will sum to zero, resulting in a complete standstill. For instance, the robot might be headed directly towards a goal point, with no walls to either side, resulting in a zero rotation vector. If the robot hits a wall straight ahead, then the basic goal-oriented forward push can balance the obstacle avoidance backward push, resulting in a zero translation vector (Figure 8). The robot can remain indefinitely in this position until some kind of force allows the robot to escape from these local minima.

In the present configuration there are two schema that are implemented to try to avoid local minima. The first, the `stuck` schema, monitors the total summed translation and rotation recommendations and determines when several consecutive cycles pass with low totaled recommendations. Consecutive low recommendations often indicate that the robot is at a local minimum. The `stuck` schema then recommends a short burst of high intensity rotations in a consistent direction, designed to turn the robot away from an obstacle in front of it and break the deadlock. There is also a schema implemented that gives a consistent level of variability to the translation and rotation recommendations, to keep a constant level of fluctuation in the motor recommendations that can help to keep the robot out of local minima in the first place. The `fluct` schema recommends a smoothly varying value for both translation and rotation. This value is tied to a time-based `sin` function, yielding a smooth and constantly changing value to be summed into the final recommendations. This gives a noise-like variability to the vectors which will not cause the jerky behavior associated with true random noise functions. The particulars of each of the schema are discussed in Appendix A.

5 Behaviors

5.1 Behavior Basics

The previous section gave a thorough description of the schemata and the parameters that can change the way they function. The schemata have been intentionally left as general as possible. In order to make these schemata perform well in the task of exploration, they need to be grouped and parametrized into the actual behaviors that will govern the function of our exploration agent. By grouping the schemata in a variety of ways and altering the parameters on these schemata, unique, specialized behaviors can be formed. A behavior might consist of a single schema, or it may be a combination of many schemata. The behaviors used for the exploration system tend to run a number of schemata at once, as there are many different aspects of navigation that need to be considered at any one time.

All behavior specifications are contained in the `control` module, which is also responsible for the function of the schemata. When an upper layer sends a command to change behaviors, `control` re-initializes the data fields containing the schemata parameters and the data field that maintains information about which schemata are currently "turned on". Each behavior definition first specifies the parameters associated with obstacle avoidance. The behavior definition then sets the parameters for each schema that will be running in the new behavior, and sets a byte in the `schema_on` array that tells the schema processor that a particular schema function should be called. When all parameters for the desired schemata are set and those schemata are "turned on," then normal schema function resumes, with the new recommendations being governed by the new parameters.

5.2 System Behaviors: The Door Dilemma

The schemata contained in this system, while generalized, were more or less specifically designed to negotiate the problems associated with exploring an unknown environment. Thus the behaviors used by HYSTE often look very similar, differing only in a few significant ways. I dubbed the first behavior `CRUISE`. It is intended to be used when the robot needs to move quickly from goal point to goal point over a relatively unobstructed area. It only uses a few of the schemata, as it is not intended to for use in the close quarters that many of the schemata are designed for. The rest of the behaviors differ in the "aggressiveness" of the motion. I hesitate to use biomimetic terminology, but it seems appropriate in this context. Much research energy was spent finding the "perfect" exploration behavior: a behavior that explored areas thoroughly while staying away from walls. The main difficulty came in squeezing through very tight spaces, spaces which might be only a little broader than the robot. Doors proved extremely difficult for the schema-based system to negotiate, in large part because of the highly inconsistent nature of the sonar sensors. One side of the door would cause a strong push until the robot moved to have its sonars facing a corner of the door, at which point the sonar ping would bounce incorrectly, causing the robot to occasionally swing violently in that direction, as it would appear for a moment that there was no obstacle in that direction. The IRs will always return a value, but their width range is slightly broader, meaning that an IR might not see the door gap at all. Finding a consistent set of sensor indications that suggested that a door was there proved as difficult as getting through the door.

The following behaviors represent an abandonment of the search for the "perfect" behavior. Instead, each contains tradeoffs of safety and speed for the width of the gap that the robot can negotiate. Any

of these behaviors can easily get the robot through a gap twice the size of the robot, but only the more aggressive behaviors can let the robot negotiate a gap only 125% of its size, or even a gap smaller than the robot that can be made larger by bumping into one of the sides. All of these behaviors are constructed to operate in close proximity to objects, and to exhibit ant-like abilities to get around objects. All exploration behaviors cause the robot to move directly towards the goal point until an obstruction is encountered. If the robot approaches an obstruction at an angle, then the obstacle avoidance schema will generally cause the robot to skirt the obstacle easily. If an obstacle is encountered head-on, then it will generally cause the robot to come basically to a halt until the local-minima avoidance `stuck` schema turns the robot to one side or another. The robot will then generally do something that looks like wall-following, as the push towards the goal is counter-balanced by the side push away from a wall. A gap to the wall-side will cause the robot to turn quickly towards the space, where the width of the gap, the level of aggression, and the exact heading to the goal point will generally determine whether or not the gap can be traversed. The parametrizations that cause a behavior to be more or less aggressive are discussed below in the specific behavior descriptions.

Native to the reactive layer is a function that adds to the utility of the `stuck` schema. The `stuck` schema is set to recommend that the robot turn in the direction specified by `stuck_direction`. The `stuck` schema was found to be more effective at helping to achieve goal points if the `stuck_direction` varied with time. Thus `stuck` would recommend that the robot turn left when judged stuck for a certain period of time, then turn right for twice that much time, and then left for twice that much time. When the behavior is specified in the reactive layer, a time stamp is initialized to a base duration value. The `stuck` schema will recommend that the robot turn right if it is judged stuck during that base duration. After that base duration, the duration is multiplied by two, and `stuck_direction` is changed to recommend left turns. This continues until a new behavior is specified, at which point all values are reinitialized.

5.2.1 `CRUISE`

As noted above, the `CRUISE` behavior is not intended to execute in dense areas, in which it will have to get through extremely tight gaps to achieve a goal point. Rather, it is meant to quickly and efficiently achieve a goal point while moving through relatively unobstructed space. However, the topological mapping as presently instantiated can guide the robot through unobstructed paths that are very close to obstructed areas. Thus the design of `CRUISE` pre-supposes that when the robot is using the `CRUISE` behavior a path exists to the goal point, although that path may be a tight fit for the robot. `CRUISE` doesn't necessarily need to be parametrized to get the robot through the tightest places however, as another behavior, `CRUISE_AGG`, is designed to move through those tight gaps. Obstacle avoidance parameters for `CRUISE` attempt to emulate this design strategy. Obstacles beyond a certain close distance are completely ignored, although obstacles that are closer than this distance exert a substantial push on the robot. This parametrization allows the robot to move near obstacles without being affected, until the obstacles are so close that they must be avoided. Sensor weightings are specified such that the front side sensors push the robot backward much less than the front sensor would if an obstacle was detected at the same distance. This allows gaps to be negotiated more easily, as the sides of a gap cause only a small backward push, allowing the robot to move through the gap if it is properly aligned. The `goal` schema is parametrized to allow little exploratory lenience, recommending a strong goal push if the goal heading is deviated from more than a low value. Finally, the local minima avoidance schemata are functional in `CRUISE`, but parametrized to function

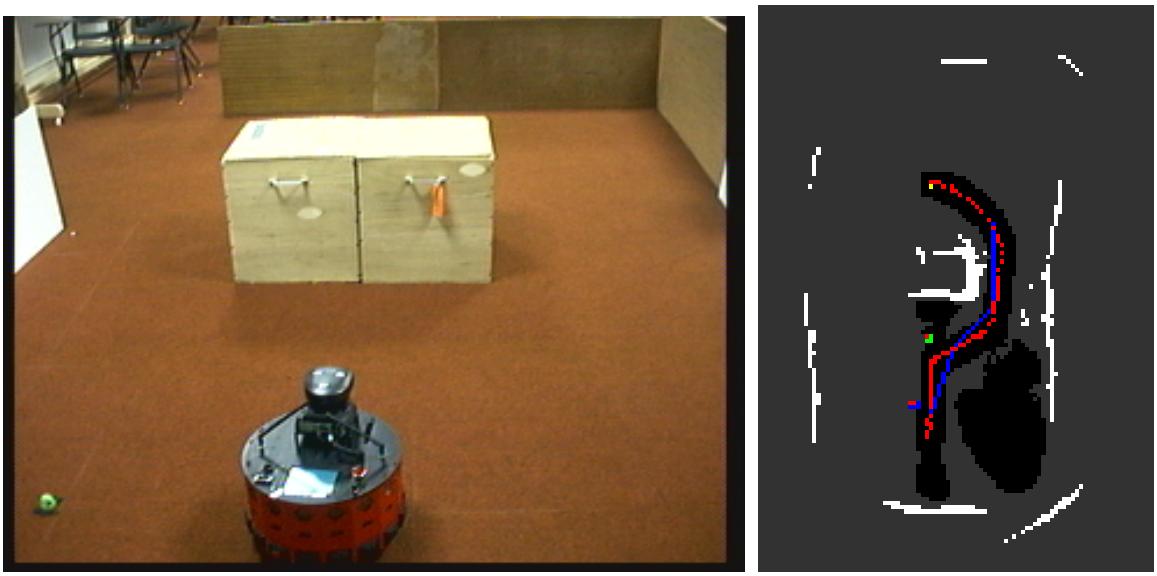

Figure 9: A simple obstacle avoidance scenario using `FAST_SAFE`. The robot neatly skirts the box on the right side to move to the goal, which lies directly beyond the box

only when something very close to a perfect standstill is achieved, as `CRUISE` should not turn away from any negotiable gap. A more complete discussion of the exact values that create this behavior appears in the behavior Appendix B.

5.2.2 `FAST_SAFE`

The first behavior designed for use in exploration near obstacles is the `FAST_SAFE` behavior. It is designed to cause the robot to move towards a goal point quickly, but safely, and explore an area cursorily if its path to an obstacle is blocked (Figure 9). Several parametrizations allow this behavior to create relatively safe and quick movement, at a cost of thoroughness. The first set of parameters specified are the weightings that different sensors receive in each direction, the `sensor_model` values. The effect of altering these weightings is discussed at length in Section A.2. By giving the sensors directly to the left and right of perfect front a high weighting relative to the front sensor, the robot will tend to stay a considerable distance from obstacles to the front. The price that is paid for this is that this behavior is much less likely to cause the robot to go through narrow gaps; the walls to the side of a gap cause almost as much of a backwards push as would occur if there was a wall completely occupying the gap. Also, the weightings associated with the side and back sensors are raised, ensuring that corners will not be taken too tightly.

Other parameters also allow for a greater measure of safety. The `care` values of this behavior are set fairly high since this behavior is intended to act in close quarters to obstacles. This means that obstacles will begin to affect the robot when it is still relatively distant from them. The `max` values are also set high relative to the goal pushes, again to ensure that the robot does not come too close

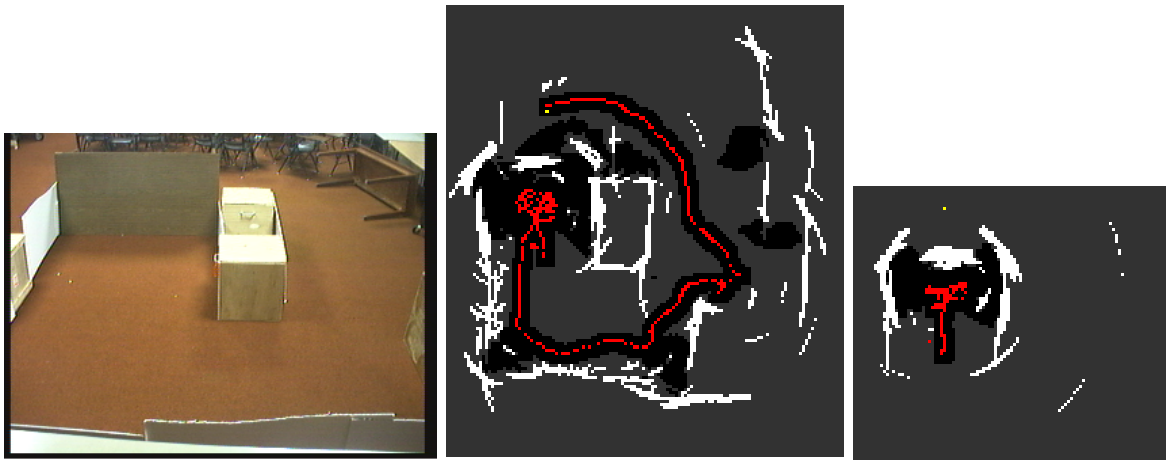

Figure 10: Left: The scenario. Middle: The robot, run with the FAST_SAFE behavior, successfully achieves a difficult goal point using the goal schema in wall-following mode. The robot started about 1.5 m in front of the table in the left side of the picture. Right: The robot with SLOW_SAFE behavior, which disables wall-following, cannot achieve this goal.

to obstacles. The bump schema is also functioning in this behavior, as it is in all behaviors.

The other schemata are parametrized to make this behavior a fast first-wave explorer. The goal schema max_angle and min_angle parameters are set fairly high to allow this behavior the leniency to explore if a direct path to the goal point is obstructed. The max_rotate parameter is set low for the same reason, and push_trans high to promote fast exploration. Additionally, goal_follow is set to one, making the robot follow walls much longer than it would otherwise. In the first moments of exploring an environment, there are few techniques that explore an area more effectively than an extended wall follow. Many complicated goal points can be achieved simply by wall following (Figure 10). Unfortunately, wall following can also lead the robot on long and time-consuming circular paths, that help explore an area initially, but are wasteful when a more thorough approach is necessary to explore additional area. FAST_SAFE is intended to be the first exploration behavior used when setting a new goal point in the deliberative layer, and as such it was decided that the extra exploration goal_follow allows was worth the possible time inefficiency. It is currently the only behavior that allows wall-following.

The stuck schema also has a role in making this behavior a fast but cursory explorer. FAST_SAFE's stuck schema is parametrized to make it recommend turns very quickly, meaning that at the slightest indication of a local minimum stuck will turn the robot. FAST_SAFE creates this stuck schema by setting both stuck_trans and stuck_steer to be relatively high, and stuck_threshold to be only a few cycles. This means that only a few consecutive cycles of low summed recommendations from the schemata are sufficient for stuck to begin recommending a turn.

Finally, the `fluct` scheme is turned on, and given fairly high values, as anything that keeps the robot using `FAST_SAFE` out of local minima will help it explore area quickly. Adding a translational fluctuation can aid in exploration while still making the behavior a quick, safe mode for exploration.

5.2.3 SLOW_SAFE

The `SLOW_SAFE` behavior is the first of the behaviors designed specifically to maneuver through doors and other tight spaces to perform the slower, more careful exploration that is required to fully explore a space. In terms of "aggressiveness" this behavior is the least aggressive of the thorough explorers, designed not to hit anything but to get through some of the tight spaces that the `FAST_SAFE` behavior will cruise past. It differs from the `FAST_SAFE` behavior in only a few respects. This behavior is basically a cross between `FAST_SAFE` and `AGG`: slower, more thorough, and more aggressive than `FAST_SAFE`, but less likely to bump into things than `AGG`. Its exact specifications can be found in Section B.0.2.

5.2.4 AGG

The `AGG` behavior is the first of what I call the truly "aggressive" behaviors, behaviors that sacrifice the robot's safety to some extent in exchange for giving the robot the capability to move through spaces not much bigger than the it. These aggressive behaviors are not designed to be exploring behaviors. Rather, they are intended for use in a situation in which a goal point appears achievable, but may require that the robot get very close to walls, or even bump them (Figure 11). They will be used by the deliberative layer only when a goal point cannot be achieved by a safer behavior, as in Figure 13.

A main determinant of the aggressiveness of a behavior is the sensor weightings held in `sensor_model`. `AGG` has much lower values for both the front side sensors and the back side sensors, making it quite capable of finding and pushing through doors and other tight spaces. Lowering these parameters does mean that the robot is more likely to rotate into obstacles, or bump them lightly. But the robot is sturdy, and bumping against some walls may be the price that must be paid for fully exploring a space. The IR threshold value remains constant in the other behaviors, but it is lowered in `AGG` behavior, meaning that the IRs are trusted less. Though they are generally more accurate at detecting obstacles extremely close to the robot, the width of their beam is slightly larger than that of the sonars, meaning they less able to detecting smallish gaps in an otherwise uniform obstacle. Thus they will only really be used at distances at which sonars begin to completely fail. `Care` values are set low, as the robot should only be affected by things that are quite close to it. The `max` values are also relatively low, as the robot should be capable of moving very close to obstacles before they cause a repulsive push. The ratio of the front `max` value and the goal translation push is especially low. In the `goal` schema, `max_angle` and `min_angle` are low, as `AGG` should make the robot goal focused. The `max_rotate` and `push_trans` parameters are set fairly high to aid in creating the aggressiveness of the behavior. The `bump` schema remains unchanged, except in this behavior it might have more to do!

The local minima avoidance parameters are set in a somewhat peculiar manner. The `stuck` schema is parametrized to be virtually non-functional, as passing through gaps often requires the robot to spend a lot of time at very low speeds worming through a gap or tight passageway. In early stages of testing there were times when the `stuck` schema actually caused the robot to turn and retreat when it was almost entirely through a gap! The `stuck` schema remains on nonetheless, as there are

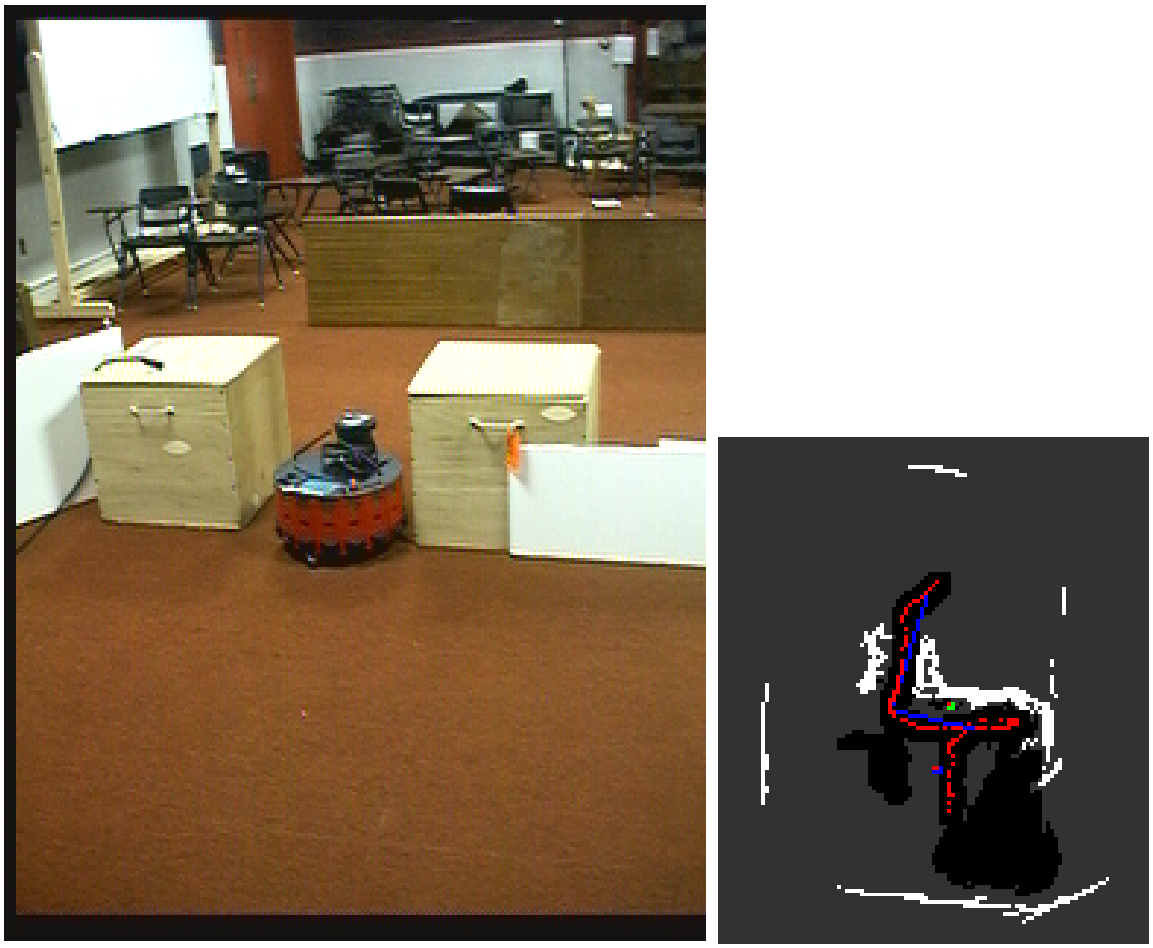

Figure 11: The robot, using `AGG` behavior, moves through a very tight gap to a goal beyond the obstruction

Figure 12: The robot using `AGG` moving through a gap to a goal point beyond the boxes. See Figure 13 for evidence grids from different behaviors in this scenario.

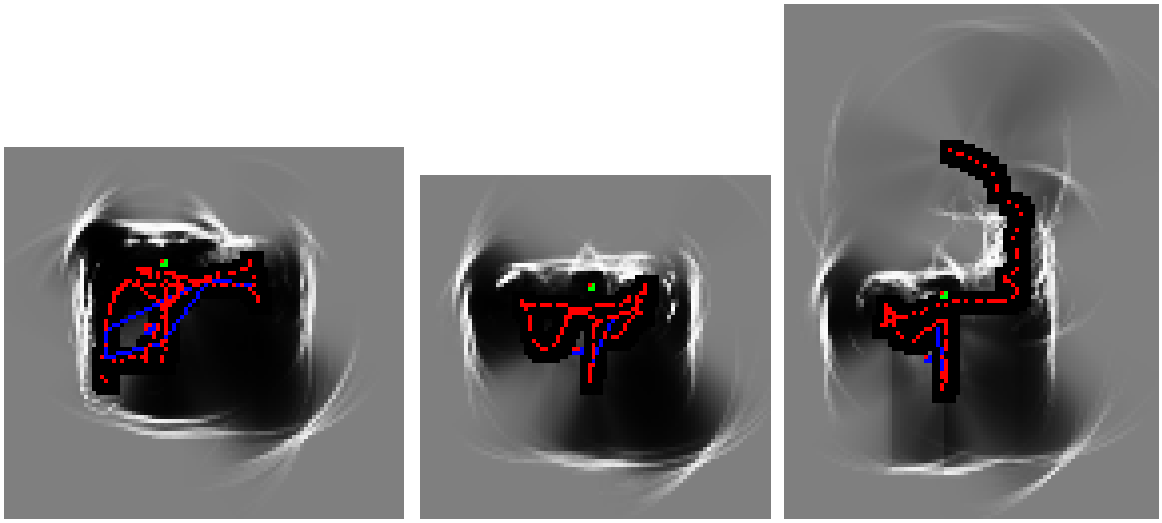

Figure 13: The robot with three different behaviors attempts the scenario shown in Figure 12. Left: The robot cannot achieve the goal point using `FAST_SAFE`. Middle: The robot using `SLOW_SAFE` has similar difficulties achieving the goal point. Right: The robot, using the `AGG` behavior, makes it through the gap only 15 centimeters larger than the robot.

times when getting out of the vicinity of a gap and making another pass is a good thing to do. The `stuck_threshold` parameter (Section A.5) is reduced substantially, as `stuck` should recommend only a brief turn unless there is a wall extremely close to the robot. The other minima avoidance behavior, `fluct` is actually turned up a lot in this behavior, more for waggling purposes (Section A.6) than local minima avoidance. A substantial front/back fluctuation is added, which can help pop the robot through a difficult gap in some situations.

5.2.5 CRUISE_AGG

`CRUISE_AGG` is a more aggressive version of `CRUISE`. `CRUISE_AGG` is intended exclusively for use in the case that `CRUISE` can't quite get the robot through a tight gap. The topological mapping will currently show a path as unobstructed if a path a single cell wide can be found through a region, even though that might constitute a passageway five times smaller than the robot. `CRUISE_AGG` should be able to get the robot through any gap that the robot can physically squeeze through, although the robot may bump the sides along the way.

5.2.6 VERY_AGG

`VERY_AGG`, as the name suggests, is a more extreme version of `AGG`. This behavior is designed to function almost as a bumping behavior, to actually enact change in the environment in order to get through gaps that are too small for the robot. If a partially open door is the only exit from a room, then the only way for a robot to get out into the world is to actually bump the door. This behavior, if supplied with a well-chosen goal point, should be able to guide the robot through an opening if it is physically possible. The front side sensors are turned down even more, to the point that the only sensor that really pushes the robot backwards is the very front one. The only other major change is that the front `max` value is lowered and the `goal` schema `push_trans` variable is raised until the two are almost equal. This means that the robot basically has to bump into a wall, initiating a strong backward push from the `bump` schema, before the robot will move away from a wall.

6 The Deliberative Layer

6.1 Overview

As the schematic of the architecture suggests (Figure 1), the deliberative layer has two main purposes. The first is to evaluate the entire evidence grid and select a goal point that appears as if exploring it will result in the exploration of unknown area. Once the deliberator layer has selected a goal point and given it to the reactive layer however, it may not be instantly achieved by the reactive layer. This does not mean that the goal point is completely unachievable. It may simply indicate that another behavior should be selected. But the deliberator should be able to differentiate between situations in which changing behaviors may help, and situations in which the selected goal point is going to be inaccessible regardless of the behavior used. The first part of this section describes the techniques used on the evidence grid to yield a likely goal point, and the methods used to get the the robot to a configuration that gives it the best chance of achieving the assigned goal point. The next portion describes the evaluation procedure used on an existing goal point to attempt to determine whether or not the goal point has continued viability. Finally, adaptive processes are described that adjust processing parameters when a likely goal point can not be found given the current set of parameters.

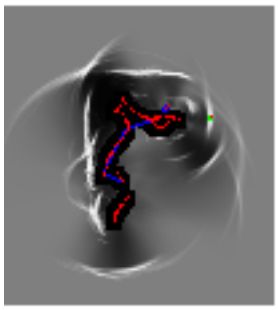

Figure 14: This is an evidence grid before thresholding. This evidence grid will be taken through all of the stages of image processing used by the deliberative layer. The more white a cell is, the more likely it is occupied. The red (or light grey) cells in the center of the black are the cells that have been occupied by the center of the robot.

These adaptive processes attempt to make the system durable even under very adverse circumstances. Results and examples of the entire system in action are in shown in Section 7.

6.2 Thresholding the Evidence Grid

The deliberator must take the capabilities of the reactive layer and the mapping information compiled by the middle layer and use them to fully explore an area. Before the deliberative layer can begin planning, it must first get access to mapping information held by `mid` (Figure 14). As this information is held in local `mid` variables, when the deliberator needs a map it must request it from the middle layer. It does this by setting a variable in the shared memory that is checked by `mid` every time cycle. When the variable is set, `mid` will call a function that both thresholds the evidence grid and converts into an image format. The deliberative module requires an image in PPM format. PPM format gives color images, where each pixel has a red(`r`), green(`g`), and blue(`b`) value. All manipulations could be performed on a more compact greyscale image, which contains only a single value per cell, but the distinctions between different cells in the resulting pictures would be less clear to a human observer. The thresholding is performed in a parametrized fashion. The deliberative module contains two parameters `occ_thresh` and `un_thresh`. `Occ_thresh` thresholds the evidence grid occupancy above which a cell is considered occupied, while `un_thresh` specifies the evidence grid occupancy under which a cell is considered unoccupied. Cells judged occupied are entered into the PPM image as white ($\texttt{r} = 255; \texttt{b} = 255; \texttt{g} = 255$), and cells judged unoccupied are given a completely black value ($\texttt{r} = 0; \texttt{b} = 0; \texttt{g} = 0$). All other values, judged unknown, are given a grey value($\texttt{r} = 50; \texttt{b} = 50; \texttt{g} = 50$). These values are entered into a PPM image variable in the planner shared memory segment (Figure 15 (left)).

Additionally, the time-stamped visitation map discussed in the introduction to the middle layer is held within `mid`. This information is also loaded into a deliberative shared memory variable by the thresholding function. When this function has finished executing, the deliberator should have access to all the information necessary to designate a new goal point. `Mid` sets a shared memory variable that informs the deliberator that the operation has concluded, and the deliberator can initiate execution.

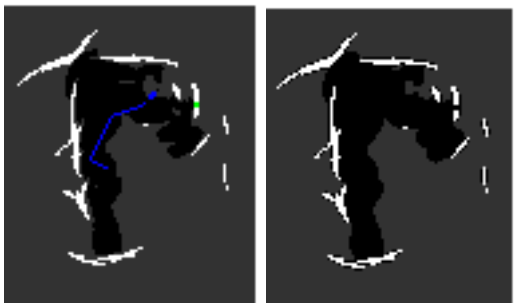

Figure 15: Left: The evidence grid directly after thresholding. Lines in the black represent the topological mapping. Right: The thresholded image after it has been shrunk with a `neighbor_thresh` value of 4.

Figure 16: The shrunk image with frontiers found in pink (or light grey). The frontier processing had an `occ_dist` value of six

6.3 Shrinking and Frontier-Finding

Once the image has undergone thresholding, HYSTE has a natural goal: to make all unknown cells become known; if every cell physically accessible to HYSTE has been correctly judged occupied or unoccupied, HYSTE has successfully explored an area. But finding a goal point which will cause unknown cells to become known quickly and effectively is a more complicated issue. The first operation performed on the image is a shrinking algorithm [15]. Evidence grids are prone to noise, and the thresholds governing which cells are judged occupied, unoccupied, and unknown were experimentally determined and therefore somewhat arbitrary. Thus there can be small pockets of unknown area surrounded by unoccupied area, or small pockets of unknown area lurking directly next to walls. Chances are that these cells do not actually represent unnoticed features, and should not be classified as frontier. The shrinking function seeks to erase these areas before frontier detection. This function examines each cell in the image. If that cell is white or black, its value is simply copied into the corresponding position in a new, shrunk image. However, if the cell is designated unknown, all eight of its neighbors (diagonals included) are examined. If fewer than an adjustable parameter `neighbor_thresh` neighboring cells are also unknown, the cell is marked on the shrunk image as unoccupied (Figure 15 (right)). `Neighbor_thresh` is currently initialized to four.

Figure 17: Left: The frontiers of the region. Right: The results of connected region extraction. The different connected regions are marked in different shades of grey, and numbered one to six.

Once unknown cells that likely to be simply noise are removed, the frontier cells are determined. The frontier-finding function examines each cell in the image except those on the border. If a cell is unknown or occupied, it simply copies the values to a new frontier image. If the cell is unoccupied, then it has a chance of being a frontier cell. A helper function examines the individual cell. It first looks at all the cell's neighbors. If any neighbor is an unknown cell, then the function examines the cells in a larger square around the original unoccupied cell. The number of cells on each side of the square is specified by a parameter `occ_dist`, which is initialized to six. If any cell in the 36×36 cell area is occupied, then the cell is not judged a frontier cell. If the cell has an unknown neighbor, and the cells in the box are all either unknown or unoccupied, the cell becomes a frontier cell, designated bright pink on the new map. Cells within a certain range of an occupied cell are excluded because they may not be accessible. The deliberator should select a goal point that lies within a just a few cells of a cell that probably contains an obstacle. Figure 16 shows the map with the selected frontier cells.

6.4 Finding the Largest Connected Region

The first method considered for deciding which frontier cell should be recommended for exploration examined each square of the image, finding the square with the greatest number of frontier cells. The cell at the center of the square with the largest number of cells became the "hot" cell, and was used in goal point selection as described below. However, this method did not seem the best possible algorithm for finding a likely frontier. A square containing ten disconnected frontier cells spread about would be more likely to be chosen than a square containing seven cells in a line. The spread-out frontier cells would be more likely to be noise, while the seven cells in a line could have been a small but negotiable gap in a wall, a true frontier. Thus it was decided that exploring the largest continuous group of frontier cells was the most likely to result in visiting previously unexplored cells. To find the largest continuos region of frontier cells a fast, two-pass connected region extractor (CRE) was written [15]. This algorithm finds the largest connected region of frontier cells, and returns the number of cells in that region as well as the cells' identities.

The strategy used by the CRE is two passes for identification followed by a final third pass to copy the identified cells of the largest region to a shared memory location. It seeks to mark each frontier cell with an ID number corresponding to the connected region to which it belongs. It is acceptable that different members of a connected region have different IDs as long as it is noted that all the region ID numbers in a connected region actually constitute a single region, and not several distinct regions.

A new data structure is created for the CRE which contains an integer variable for each cell in the image. This structure is used to hold assigned ID numbers. On the first pass, the CRE examines each cell in the frontier image. If a cell c has been designated a frontier cell, all of its neighbors are examined. If none of c's neighbors has been assigned a region ID, then a new, unique region ID is assigned to c and the next cell is examined. If a single neighbor has already been assigned a region ID, or multiple neighbors have been assigned IDs, but all neighbors have been assigned the same ID, then c is also assigned that ID and the next cell is examined. If c has neighbors that have been assigned distinct IDs, additional steps must be taken. These neighboring cells were labelled with distinct IDs, suggesting that they were in different regions. However, a cell, c, was discovered that borders on all those cells, so those distinct ID numbers must actually designate the same region. A table containing these region equivalences is maintained, to hold the information indicating which region IDs actually represent the same connected region. Initially, each entry in this table was initialized to its index value. When a cell like c is discovered, it is assigned the lowest value of all of the distinct IDs of its neighbors. Then all of the neighbors with region IDs that are higher than this lowest ID have their entries in the equivalency table updated with the lowest ID. This marks that all of those regions are actually the same region, represented by the lowest ID number. Any region that has an entry in the equivalency table that is not its index must have been found equivalent to a region with a lower ID number. Once each frontier cell in the image has been found and examined, the first pass is complete.

After the first pass, every frontier cell in the image now has a region ID associated with it. There is also an equivalency table containing all the information necessary to determine which disparate region IDs have been assigned to cells in the same region. Before the second pass through the image equivalencies must be computed. Say for instance that region 5 has a 4 in its index in the equivalency graph. This means that all cells marked with a 4 or a 5 region ID belong to the same region. But region 4 may have a 2 in its index in the equivalency table. This means that all region 5 cells actually belong in region 2. Thus every entry in the equivalency table up to the number of region IDs assigned is examined. If an entry equals its index, it is left alone. If an entry e_0 does not equal its index, the CRE looks at the entry at index e_0. If the entry at index e_0, e_1, does not equal its index, the entry at index e_1 is examined. This process repeats until an e_i is found such that the entry at index $e_i = e_i$. When e_i is found, it is assigned to index e_0, and all other indices that were traversed finding e_i. When all entries are examined, the equivalency graph will contain only entries that equal their indices, or entries that indicate entries that equal their indices. Thus the entry of every region ID contains the lowest region ID of any cell in that region.

At this point, a simple second pass is made through the image. A counter table keeps track of how many members are in each of the connected regions. Whenever a frontier cell is found, the counter in the counter table corresponding to the cell's entry in the equivalency table is incremented. Thus only region IDs that are the lowest of any region ID assigned to any cell in that connected region will have a non-zero counter value (Figure 17). The index with the highest counter value is selected as the region ID representing the most members, the largest connected region. A final pass is made through the image, filling an array in the shared memory that contains all cells that are members of the largest connected group. This information is used if no members of this connected group can reach a visited cell by breadth-first search, as discussed below. If there are fewer than 20 members of the largest connected group, they are all put into a representative cell array in the shared memory. If there are more than 20 cells in the largest region, then 20 cells are chosen randomly and put into the

representative array. Finally, the number of cells in the largest region is put into a shared memory variable, for use as described in the next section.

6.5 Evaluating `Trapped` and Selecting a Hot Cell

At this point the planner has a representative number of cells in the largest connected region (LCR) of frontier cells as well as the size of the LCR. The planner next compares the size of the LCR to a parameter `trapped_fron`. If the LCR is smaller than `trapped_fron`, this could indicate that the robot is are trapped and need to take aggressive steps to find new territory. `Trapped` is set to one, and the image is reprocessed according to methods described in Section 6.9.

If there are an ample number of frontier cells in the largest connected region, one must be selected to function as the "hot" cell, a cell that is judged to mark a pathway to a frontier. The goal in selecting the hot cell is to mark a cell that the deliberator can send the reactive layer through to explore unexplored area. A breadth-first search is performed on each of the cells in the representative portion of the largest connected region. This breadth-first search is along all cells, and attempts to find the distance to the nearest occupied cell. Each representative cell is given a distance to the nearest occupied cell and the cell with the greatest distance is selected as the hot cell. The largest connected region may be just a small line of cells with occupied cells on either side, indicating a door. Finding doors, and assigning goal points that will get the robot through the door, are primary goals of HYSTE. Thus the deliberator should pick the cell that is in the exact middle of the door, and try to send the robot through that, rather than choose a cell close to either side of the door. In situations in which the door might be large, perhaps 30 cells in width, picking the exact center of the connected region is probably unnecessary. The same is true for largest regions containing a hundred or more cells, which likely indicate empty space that can be explored by picking any point in the region. These considerations, combined with the computational expensiveness of breadth-first search, serve to justify the choice to evaluate only a representative portion of the largest connected region if it is very large.

6.6 Finding Nearest Visited Cell and Line Projection

Once the deliberator has found a likely hot cell, it must pick the likeliest avenue of approach for moving through that hot cell. Attempting to move through this hot cell from beyond a wall, for instance, would probably not result in exploring the area around the hot cell. The methods used here place a high value on cells the robot has already visited. Because the environment HYSTE is exploring is expected to be relatively static, the robot should be able to get back to any cell that it has already visited. Thus if the robot has visited a cell, and the deliberator can reach the selected hot point by a breadth-first search along unoccupied area, then the hot point has a decent chance of being reachable from the visited point. Additionally, the topological mapping supplied by the middle layer can be used to give an actual sequence of goal points that can take the robot from its current location to any previously-visited cell. As such, a breadth-first search along unoccupied area with unoccupied neighbors is conducted from the selected hot cell. It was found during experimentation that considering any unoccupied cell in the BFS would allow paths of a single unoccupied cell in width, which is almost certainly too small for the robot to fit through. By only considering cells with all unoccupied neighbors, if the BFS from the hot cell can find a previously-visited cell, a path at least three cells wide from that previously-visited cell to the hot cell exists; a path three cells

Figure 18: The final goal selection image. The nearest visited cell is shown in red (or is slightly darker than the frontier pixels), with the "hot" cell slightly lighter than the nearest visited cell. The projected line shows up as blue (or darkish grey), and the actual selected goal point is an orange (or lighter grey) cell at the end of the projected line.

wide has a good chance of turning out to be negotiable. This BFS attempts to find a cell that has already been visited by the center of the robot. The information about the visited cells is held in the time-stamped visitation array. If the BFS is unsuccessful, chances are that the frontier cell is actually inaccessible from all currently explored avenues of attack, and its unoccupied status was the result of erroneous sensor readings. In this case, all cells in the largest connected region, which were stored in a large shared memory array, are reassigned as unoccupied cells, and the process of finding a hot cell is repeated on the next largest connected region.

If a previously-visited cell is found in the breadth-first search, the deliberator has selected both a likely frontier cell and an avenue of approach to that cell. To attack this frontier by the selected avenue of approach, the robot must be guided to the nearest-visited cell, and then can attempt to move through the frontier represented by the hot cell. But if the hot cell is selected as the goal point, while that single cell may be explored, nonetheless the whole idea of a frontier is that it may represent passage to even more unexplored area. Thus ideally the robot should move through the hot cell into the unknown area beyond. Selecting a goal point beyond the hot cell could yield a completely unattainable goal point; it could very well be in the middle of a wall. But the strength of the reactive layer as is is formulated in HYSTE is that it will is that it will explore a number of approaches to attaining a goal point. If the the hot cell is set as the goal point the robot may have a better chance of achieving the goal point quickly, but may ultimately fail to explore an area as thoroughly or efficiently as setting a goal point in the unknown beyond a frontier may permit.

Thus the deliberator needs to maintain the same avenue of approach to the hot cell, but continue it beyond the hot cell into the unknown beyond the frontier. A line projection is performed using the nearest-visited cell and the hot cell as guide points. The line begins at the nearest-visited cell, and is projected through the hot cell into the unknown beyond the frontier for a certain distance. The slope of this line is computed using the nearest-visited cell and the hot cell. The slope information is used to draw a one-cell wide line from the nearest-visited cell, through the hot cell, continuing along the same slope for a parameter `line_dist` number of cells, currently initialized to 15. The final cell in this line becomes the goal point recommended by the deliberative module (Figure 18).

6.7 Finishing Up and Achieving the Nearest Visited Cell

The deliberator now has a goal point and a likely avenue of approach to that goal point represented by the nearest-visited cell. These are not in a form that utilizes the reactive layer in the correct manner, as the goal cell may lie several rooms away from the nearest visited cell. Thus merely setting that cell as the next goal point for the reactive layer will not give the robot a good chance of achieving the goal point. First, the robot should be guided the robot to the nearest-visited cell, and then the reactive layer should be assigned the selected goal point once at the that cell. Fortunately, `mid` contains a function that will return a sequence of goal points that will guide the robot from any visited point to any other visited point. The deliberator sets a variable in the shared memory, which communicates to `mid` that goal point sequencing is required. `Mid` reads the nearest-visited cell in the deliberator's shared memory segment and finds a sequence of goal points that will guide the robot from the current position to the nearest-visited cell. This process is discussed in more depth in the introduction (Section 1.2). `Mid` places the sequence of goal points in a deliberator shared memory array and communicates to the deliberator that the information is ready.

A goal point has been selected, and the deliberator enters a different mode. It sets the reactive behavior to `CRUISE` and delivers the first goal point in the sequence. If the information in the topological graph is accurate, the reactive layer should achieve this goal point quickly and without encountering major obstructions. If the topological data is incorrect, and there really are significant obstacles along the path, or if the path goes through a small gap, the robot, using the `CRUISE` behavior, may fail to achieve the goal point. Currently, the deliberative module allots the amount of time in the standard planning duration for the robot to achieve the next goal point in the sequence, but it is intended that a variable amount of time based on the distance to the next point will be allotted. If the robot has not reached the goal point in the allotted time, the behavior is changed to `CRUISE_AGG`, and the the robot is given a certain amount of time to achieve the goal point with that behavior. `CRUISE_AGG` is parametrized to cause the robot to more thoroughly and fearlessly move towards the goal point, ignoring obstacles along the way, unless adjustment is absolutely necessary. Unless the topological representation has very incorrect information, the robot using `CRUISE_AGG` behavior should be able to achieve the goal point in a short time, though it may get very close or even bump obstacles along the way. If the robot using `CRUISE_AGG` can't even reach the goal point, then a new goal point is picked, as chances are likely that the new information will have been discovered while trying to move to the obstructed goal point. But generally speaking, it is assumed that the topological information is correct enough that the nearest visited cell can be achieved without major difficulty. If the reactive layer achieves the goal point in the allotted time, then the next goal point in the sequence supplied by `mid` is given, and the process repeats until the nearest visited cell is reached.

6.8 From Nearest-Visited to the New Frontier Goal

Generally speaking however, it is expected that the sequence of goal points supplied by the middle layer will guide the robot to the nearest previously visited point without difficulty. Once at this cell, the deliberative layer shifts gears yet again. There is no guarantee that the segment from the nearest-visited cell to the selected goal point is even possible to achieve, much less achieve quickly. The deliberator passes the selected goal point to the reactive module, sets the behavior to the exploring `FAST_SAFE` behavior, and time-stamps the assignment of the new goal point.

The goal point may be achieved quickly, in which case the whole evaluation process repeats and a completely different goal point is selected. In many cases though, the goal point will not be so easily achieved. If the goal point is not quickly achieved all may not be lost. The ultimate goal of the deliberator is to explore new area, and HYSTE is designed so that this goal can be furthered even if a particular goal point cannot be achieved. Thus the deliberator waits for a set interval, only monitoring the variable the reactive layer will set if it has achieved the set goal point. If the interval expires and the reactive layer has not reported that the goal was achieved, the deliberator first looks at the number of new cells that have been explored in the time period. This number can be obtained by requesting the most current visitation information from `mid` and counting the number of cells that have been time-stamped between the current time and the last deliberator time-stamp, indicating that they were first visited during this last cycle. If that number of newly visited cells is over a set threshold, the deliberator does not even bother to evaluate the goal point for viability. The current goal point and behavior have caused the robot to explore a substantial amount of new area, and as such the deliberator is not particularly concerned that the goal point has not been reached. The goal of the system, after all, is to explore unexplored area, and if a particular goal point and behavior are achieving this goal, the reactive layer should not be tampered with. The deliberator updates the time-stamp to the current time and maintains the goal point and behavior. This process continues until the number of newly explored cells drops below a certain threshold, signalling that the current goal point/behavior pairing is no longer effective at exploring new area and that one or both should be altered.

The goal point was chosen because it seemed a likely place to explore unexplored territory. If the current goal point/behavior pairing is no longer causing the robot exploring new area, and the goal point has not been achieved, then there are two possibilities. One is that the goal point could be achieved by changing behaviors and allowing the robot with a new behavior to attempt to achieve the existing goal point. The other possibility is that the goal point is physically unachievable; for instance, it may lie behind a solid wall. The deliberator should be able to differentiate between these two scenarios, selecting a new goal point if the goal seems unachievable, or selecting a more aggressive, thorough behavior if that may help achieve a goal point. The robot should attain any goal point that it reasonably can, as that goal point may mark a door or lead through a tight space to a large amount of unexplored area. To attempt to differentiate between these two scenarios, the deliberator evaluates the goal point for continued viability.

The evaluation process is quite similar to the the original goal point selection. The major difference is that the original goal point functions were performed on the whole image, as the whole image needed to be evaluated, whereas the evaluation for the continued viability of a goal point is performed on a specific area of an image. A box with sides of a size specified by `eval_box`, currently initialized to 10, is drawn around the old hot cell, not the goal point. This area is shrunk with a `neighbor_thresh` of two, and the frontier cells in the box determined with a `occ_dist` of four. These slightly more lenient parameters are designed to ensure that the goal point will be maintained if there is any hope of success. As has been frequently noted, moving through doors and other tight spaces is a particular goal of HYSTE. If a goal point has been set beyond a door that was too narrow for the `FAST_SAFE` behavior to achieve, the deliberator must make absolutely sure that this particular goal point is maintained and attempted by a behavior more adept at getting through gaps. Thus if there is any chance that the region around the old hot cell marks a door, exploration of that area should be continued with a

more aggressive behavior.

After the frontier cells have been determined, the connected region extractor finds the largest connected region that exists within the box. If there is a door around the old hot cell, it should be marked with a connected frontier at least several cells wide. If the largest connected region in the box is below a certain low `eval_con` value, currently initialized to eight, the goal point is determined to be no longer viable, and a new goal point is selected. If the largest connected region in the box is larger than `eval_con`, the goal is judged to have continued viability. In this case, the deliberator selects a more thorough, aggressive behavior `SLOW_SAFE`, which will have a better chance of causing the robot to move through a tight space than the `FAST_SAFE` behavior did. The reactive layer with the behavior set to `FAST_SAFE` has a tendency to cause the robot to wander away from a goal point, and thus to give the robot with the new behavior every opportunity to achieve the goal point the heading to the goal is corrected to zero using a slightly modified version of the `goal` schema, added specifically for this purpose. This type of `goal` schema gives a very strong rotation push and no translation push until the robot rotates to face the goal. Then normal function resumes. Thus even if the robot is a considerable distance away from the goal point, its heading is correct and it can begin moving towards the goal point.

If the goal point is still not achieved after the a time interval, and few new cells were visited, then the goal is again evaluated by the procedure in the preceding paragraph. If the goal is still determine to be viable, the deliberator lets the robot using `AGG` behavior have a go at it. If the robot with `AGG` cannot achieve the goal point, the system judges the goal point unattainable at least for the moment and will select a new goal point over the entire image. Thus the total time that the robot will spend trying to pursue a goal point when new cells are not being visited is $3 * \texttt{eval\_period}$, one period for each of the behaviors `FAST_SAFE`, `SLOW_SAFE`, and `AGG`. And this much time will only be spent only if the original goal point continues to seem viable after repeated attempts are made to explore the region leading up to it. This system may seem cumbersome and slow, but consider a situation in which there are many rooms with only small doors between them. If a likely frontier is found, and it doesn't seem like it is actually a wall after the robot spends considerable time trying to move through it, then it is probably a door that could be a way to a large number of unvisited cells. HYSTE must be designed to be persistent in situations where there may be a passable but difficult-to-navigate area that leads to an unexplored area. This deliberator in HYSTE will encounter the most difficulty when dealing with areas that do not resolve well to solid walls or unexplored area, like areas with chair legs or other small and difficult to detect obstacles. Methods for dealing with these situations are included in Section 8.

6.9 `Trapped`

There may be times when the methods applied above fail to find a likely frontier region to explore. the frontier cell selection process attempts to weed out cells that are too close to obstacles to be visited by the robot, but chances are that a single frontier cell does not represent a way to an unexplored region. But HYSTE is designed specifically to explore the maximum possible area. Thus if a point is reached where the deliberative processes do not find a connected region larger than several cells, different applications may wish to take different tacts. Some applications may be content with this level of exploration, but other applications may wish to truly make certain that any possible frontier

has been explored, even those that may require the robot to intentionally alter the environment in order to squeeze through a tight space. The capabilities for this aggressive exploration are available in the system.

The base parameters for the deliberative system were chosen in order to allow thorough exploration without putting the robot at too much risk. The `AGG` behavior combined with a goal lying behind a tight door may cause the robot to brush against the sides fairly hard in the process of getting through the door, but it will not cause the robot to run head into an obstacle, which may be the best way to get through a space smaller than the robot. But manipulating the parameters of the image processing functions in the deliberative module can cause different goal recommendations to be made. It is recommended that when the original parameters reach a point in which a connected region larger than four or five cells can not be found, the image processing parameters should be changed rather that selecting goal points in these little connected regions. Manipulating the deliberative parameters may yield a better recommendation of a likely frontier for aggressive exploration.

There are three main parameters that can be adjusted to make the system more liberal in finding viable frontiers. The first parameters that can be altered are the parameters that govern the thresholding of the evidence grid data. By raising both the evidence grid probability necessary for classifying a cell as unoccupied and occupied, the number of unoccupied cells is effectively increased and the number of occupied cells reduced. The danger of this strategy is that goal points and even frontier cells can be selected that actually may actually have an obstacle in them. But the initial thresholding probabilities are experimentally determined, and do not represent perfect values. Raising these probabilities is an effective way to open up some likely, if risky, frontiers. The next sets of parameters that can be changed are the ones that shrink the image. The base parameters specify that an unknown cell must have four unknown neighbors to remain unknown. By raising this value, some border unknown cells will be replaced with unoccupied cells, which may yield a better frontier in some situations. Finally, the parameters that govern the classification of frontier cells can be altered to give more frontier cells as well. By shrinking the box around an unknown cell that is checked for occupied cells, cells closer to walls that still may be negotiable are classified frontier cells.

By altering these parameters, the largest connected region found can be increased, and that largest region may represent a more likely frontier than a smaller region in the image processed using the initial parameters. The current version of HYSTE will modify the parameters a slightly, and find the largest region using the modified parameters, finding a hot cell and the nearest visited cell using the same techniques as discussed above. But instead of starting out with the `FAST_SAFE` behavior, the behavior is immediately set to `SLOW_SAFE`. `FAST_SAFE` is not designed to be an aggressive behavior for maneuvering the robot through tight spaces to a goal, and will likely be useless under `trapped` conditions. `SLOW_SAFE` is therefore a more likely first avenue of attack. If the robot using the `SLOW_SAFE` behavior does not visit new cells or achieve the goal point, then the behavior can be changed to `AGG` if the goal point still looks potentially achievable. If the robot is particularly hardy, `VERY_AGG` behavior can be used, which should get the robot to achieve a goal point even if that goal point requires pushing an obstacle out of the way. This behavior is not for use by the faint of heart however, as the robot may bump obstacles with a good deal of force.

`Trapped` behavior may be useful for forcing a way through a door or other tight space. But after the door has been negotiated, it may open up additional possibilities for easy, safe exploration. Thus `trapped` behavior only functions for the duration of a single goal point achievement sequence of behaviors. The next time the deliberative layer picks a goal point, it will attempt to so it on a an image processed with base parameters. If this processing does not yield a sufficiently large connected region, the image will be processed with `trapped` parameters. But it is hoped that setting a single goal point using `trapped` parameters and a more aggressive behavioral sequence will discover a new frontier that can be explored safely with the more exploration-based sequence of behaviors.

7 HYSTE: Experiments and Examples

7.1 A Note on the Images

The scenario images show three distinct stages of evidence grid processing. The first type of pictures shows the raw evidence grid data prior to any processing. These pictures also contain scattered red (or light grey) pixels marking all places that the robot has visited, and blue (or darker grey) lines drawn between the topological nodes. The second type of images are those produced by the deliberative goal point selector. These are thresholded evidence grids, with all frontier cells marked in pink (or light grey). All previous goal point selections are shown on the evidence grid as single orange (or medium grey) pixels. The hot cell is a single green pixel (not distinct from frontier pixels in black and white), and the nearest- visited cell is marked as a single red pixel. There is a blue (or dark grey) line projecting from the nearest-visited cell through the hot cell to the selected goal point, marked in orange (or light grey) at the end of the projected line. The third type of image is also thresholded. It contains both topological lines and the sequence of goal points created by `mid` to guide the robot from the current pixel to the nearest visited pixel and finally to the selected goal point. The topological lines are in blue (dark grey), and the sequence lines are in red (lighter grey). The current position is marked on one end of the sequence lines as a single green pixel.

Unfortunately, at the time these tests were conducted localization was not integrated into HYSTE, meaning that dead-reckoning was the only functioning form of localization. The wheel encoders on the Magellans are very good, but no wheel encoders are perfect, and imprecisions accumulate. The skew in these images is largely produced by the lack of localization.

7.2 The Scenarios

7.2.1 The First Scenario

The first scenario area (Figure 19) is a sizable room with a box in the middle and a single narrow exit to the left. The robot was successfully able to move through the exit after exploring the rest of the area. During the course of this scenario, the deliberative module set only six goal points. Much of the area was explored as robot, using the `FAST_SAFE` behavior, completely circumnavigated the box in the center twice while wall-following. This resolved the area until only a few goal points needed to be set to resolve the rest of the area. The deliberative module was able to locate the exit as a new frontier only after resorting to trapped image processing parameters (Figure 20 (middle)). The deliberative layer was able to select a goal point using these `trapped` parameters, and the robot, using

Figure 19: The first unexplored area. The robot begins exploration approximately a meter in front of the large table at the top of the picture.

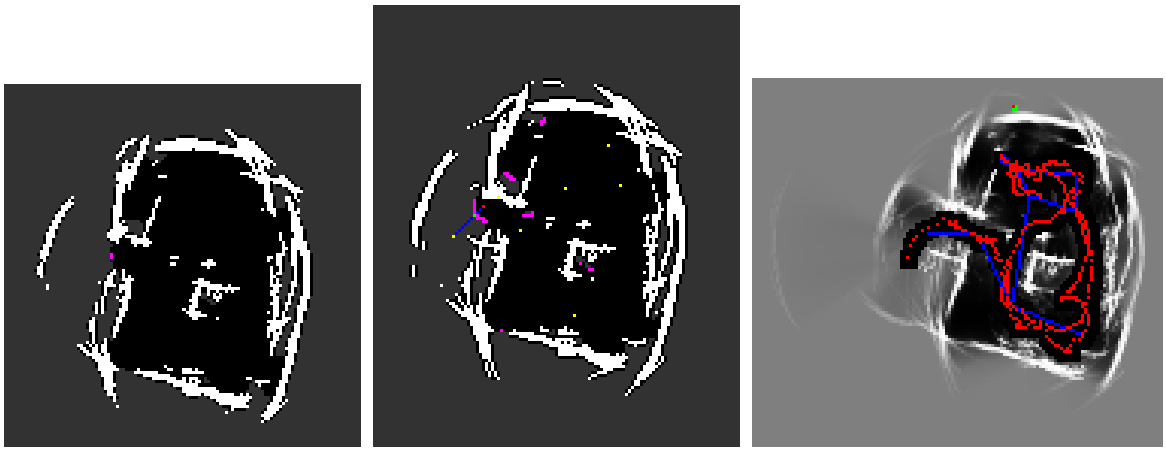

Figure 20: All pictures display the results of the scenario shown in Figure 19. Left: After substantial exploration, evaluation yields a largest connected region that may be too small to allow the robot to pass through it. Middle: The same evidence grid is thresholded according to `trapped` parameters and a goal point set. Right: That goal point, along with an aggressive behavior `AGG`, causes the robot to move through the gap.

Figure 21: A second unexplored area. The robot begins exploration approximately a meter in front of the large table at the top of the picture. Note the gap to the right side of the picture and the hallway to the left side of the picture.

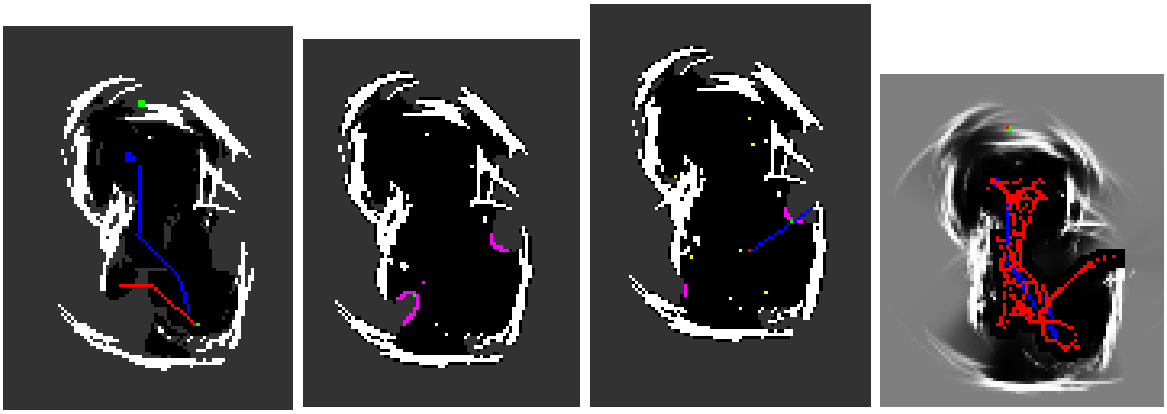

Figure 22: All pictures display the results of a first run in the scenario shown in Figure 21. Leftmost: The deliberative module successfully selects a goal that will result in the exploration of new area. Middle Left: The evaluation of a goal point (green) near the bottom of the image finds a number of frontier cells near the goal point; the goal point is maintained. Middle Right: After a brief exploration, the deliberative module selects a goal that causes the robot to move out through the gap. The long hallway is never explored. Rightmost: The final evidence grid

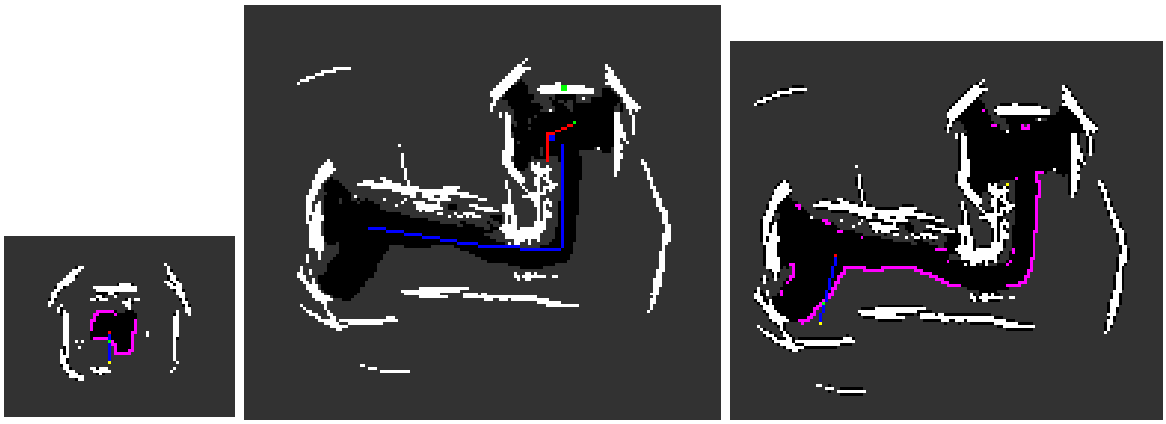

Figure 23: All pictures display a second run on the scenario shown in Figure 21. Left: After only 15 seconds of exploration, the deliberative layer selects an initial goal point. Middle: The goal point was not achievable by the robot using `FAST_SAFE`, but `FAST_SAFE` causes the robot to follow walls through a substantial amount of new area. Right: When the robot, using `FAST_SAFE`, finally stops exploring new area in the wall follow, a new goal is selected

Figure 24: More results from the run begun in Figure 23. Left: When the hallway has been completely explored, a goal point is set that requires using Dijkstra's algorithm on the topological mapping to move back through known space. Middle: A goal point is set in the final frontier. Right: The final evidence grid

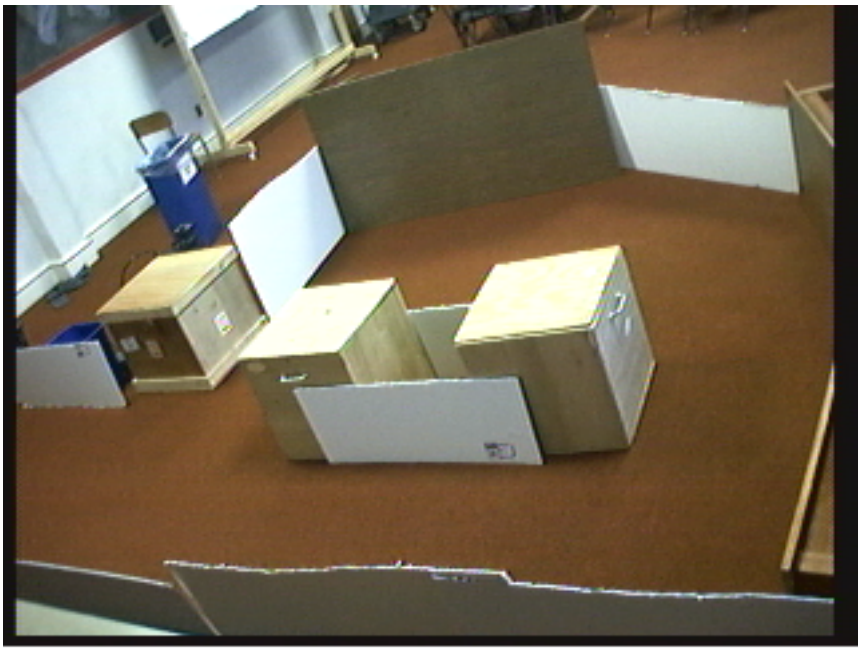

Figure 25: A third scenario, with the same starting position as in the previous two scenarios.

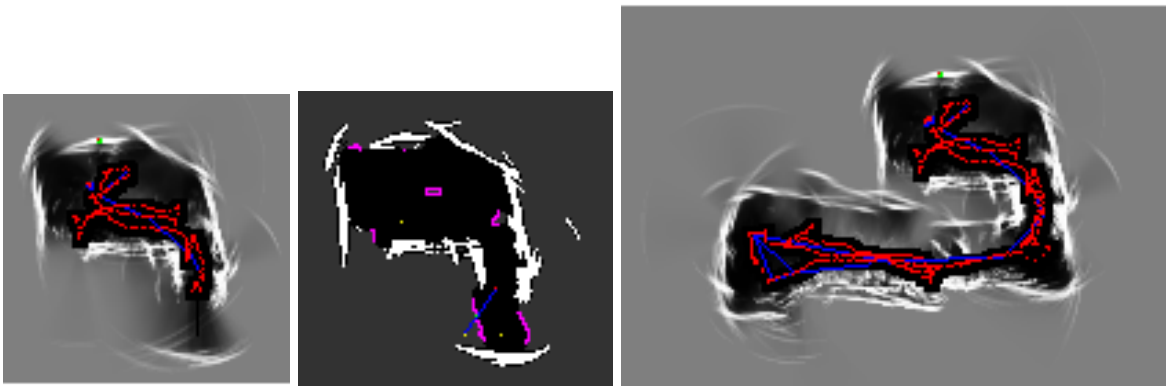

Figure 26: All pictures display a run on the scenario shown in Figure 25. Left: After a brief period of exploration, the robot manages to get through a gap into a new region. Middle: Once the new area is breached, the deliberative module quickly selects goal points to explore the new frontiers. Right: A few more goal point selections result in a thorough exploration of the area.

AGG, was easily able to move through the passageway once a suitable goal point was selected (Figure 20 (right)).

7.2.2 The Second Scenario: First Run

In this scenario there was a central room with a long hallway leading off from the left and a small gap on the right (Figure 21). In the first run conducted on this area, the robot quickly explored the area around the start position, and moved into the main portion of the central room. A goal point towards the bottom left side of the central room brought the robot near the hallway but did not cause it to enter it. The robot explored the right bottom corner of the center room before finally moving out the gap at the right of the room (Figure 22).

7.2.3 The Second Scenario: Second Run

Another run was performed in the same area, with very different results. The deliberative module layer set as its first goal point recommendation a point that was too near an obstacle to be achieved easily by the robot using the FAST_SAFE behavior. The robot was pushed around the goal point by an obstacle, and continued past the obstacle along the right wall of the central room (Figure 23). It turned into the hallway because of the wall-following behavior and continued along the length of the hallway until it reached the hallway's end. The robot made a final turn against the door at the end of the hallway, eventually turning around. After the robot turned around, it began travelling back down the hallway, exploring no new cells. This caused the deliberative layer to select a new goal point that caused the robot to further explored the small cavity at the end of the hallway. Eventually, exploration of the hallway was complete, and the deliberative layer selected a goal point back in the central room. The robot was led back through to that room using a sequence of points provided by the mid module (Figure 24 (middle)), and began exploring the central room. By this time, the robot was hopelessly unlocalized, meaning that its current estimation of its (x, y, θ) was incorrect by a substantial margin. For this reason the test was terminated before the robot could move through the gap to the right of the central room.

7.2.4 The Third Scenario

The third and final test was performed in the area shown in Figure 25. The area consisted of a small room with a large obstacle with two gaps on either side preventing easy access into a long hallway. The robot explored the central area for a short while, until the SLOW_SAFE behavior managed to cause the robot to through the gap towards the right of the obstacle although the goal point was set towards the left of the obstacle (Figure 26 (left)). Once the robot was through the gap, exploring the rest of the the hallway was unproblematic. The test was terminated due to lack of localization before the gap to the left of the obstacle was explored.

7.3 Scenario Discussion

The runs on these scenarios were all fairly successful, exploring the areas thoroughly and negotiating difficult areas that led to unexplored area. But the tests also highlight some of the main difficulties that HYSTE faces.

Open space can be negotiated easily and fairly quickly by a robot doing an obstacle-avoiding wander. In a wander, the robot moves forward unless prevented from doing do. When encountering an obstacle hindering forward movement, the robot turns in a random direction and again attempts to move forward. This type of behavior performs quite well in relatively open areas, but eventually fails when dealing with difficult areas like small gaps. To truly explore all regions of an area that are physically accessible it is necessary to generate goal points; a random wander is not enough. But assigning a goal to robot navigation, especially when using a potential fields approach, raises a host of complicated issues. HYSTE is basically designed to wall-follow until all easily reachable regions of an area have been roughly mapped. Much of the rest of the design of HYSTE is centered around determining where the difficult but passable regions are, and causing the robot to explore those regions. The examples of the performance of the reactive layer working alone (Figures 7, 9, 11, and 13) to achieve goal points shows that very tight gaps can be negotiated if a goal point is properly assigned and an aggressive behavior is designated. But the reactive parametrizations that make aggressive behaviors successful at coping with gaps do not create behavioral characteristics that lend themselves well to exploring large areas quickly. Thus HYSTE is designed to quickly explore as much area as possible, and resort to more aggressive behaviors only when the fast behaviors can no longer cause the robot to explore new area.

The example scenarios show that the methods employed by HYSTE are effective in these fairly simple scenarios. But to perform at the peak of effectiveness, moving through very tight spaces to explore as much as possible, the deliberative layer must perceive that those spaces exist. I see this as the single greatest issue in the effectiveness of HYSTE. Bluntly stated, sonars and IRs are not great sensors. The IRs accuracy range is about 25 centimeters, about the width of the robot. The sonars have a longer range, but bounce poorly off of textured surfaces, metal, and corners. Unfortunately, all unreturned sonar readings are not entered into the evidence grid. A sonar ping could fail to return because there was no obstacle within the range of the sonar, but it is also possible that the sonar ping bounced poorly off a surface. In other words, a sonar ping not returned from an obstacle 30 centimeters away looks exactly the same as a sonar ping that echoed across a wide open room. In the confined spaces on which HYSTE is designed to operate, there are far more instances of an unreturned sonar ping because of a poor bounce than because there was no obstacle in range. Inserting all unreturned sonar pings as indications that there was no obstacle in the sonar range completely ruined the evidence grids, and thus all these readings are discarded. All these inconsistencies make surviving in the world with only sonar and IR sensors a very difficult task for an exploration system. HYSTE is designed to deal with the inadequacies of the sensors. Sensor weightings attempt to compensate for the times when the most important sensors fail to get an accurate reading, and the more conservative behaviors will only move forward if all three front sonar sensors to do not sense that something is close. The price that is paid for this is conservatism and uncertainty.

HYSTE depends on the notion that exploring an area further will resolve it into known or unknown, but there are areas for which this simply not true. Corners of rooms, for instance, can show up as doors. The robot attempts to move through the corner, and is stopped by the redundant nature of the reactive sensor weightings. But the angles of the corner are such that for every sonar reading that returns an obstacle, two fail to do so. Thus the more time the robot spends in a corner, the more the corner starts to look like a door. The opposite problem can occur for actual doors. The robot can be trying to move through a tight door, when only one sonar is exactly facing the door at once, and

two others are facing the walls to the sides of the door. The sonar hits from the walls to the sides of the doors can slowly fill in the gap. If there was an obstacle beyond the gap that was within sensing range, then those distant readings will generally preserve the gap. In scenario one (Figures 19), the gap was preserved enough to be found with `trapped` parameters, but largely because there was an obstacle a short distance back that returned sonar pings. But picture a scenario in which there was no obstacle to return sonar pings within the range of the sensors. The sonars from either side of the goal would slowly narrow the gap in the evidence grid, and every ping that actually was sent through the gap would not be returned and would be discarded as noise. The deliberative layer cannot set goal points that will intentionally move the robot through gaps to unexplored area if those gaps look no different than walls.

Currently, HYSTE can generally sense gaps of a width of about 70 centimeters if there is an obstacle within the sensing range through the door. But very small gaps, or larger gaps that have no obstacle within sensing range beyond the door are quite difficult to sense no matter what image-processing techniques are used on the evidence grids. The hall in the first run of scenario two was quite large, about 1.5 m wide, but it was poorly sensed despite the fact that the robot spent some time quite near it (Figure 22). If the robot had spent more time by the gap, the gap might have resolved enough that the deliberator would have set a goal point that would have caused the robot to explore it. In this particular scenario, the deliberator was largely unnecessary, as the gap was wide enough and set up in such a way that the robot, using the `FAST_SAFE` behavior, would have eventually wall-followed through it. This wall-following occurred in the second run (Figures 23 and 24) despite the fact that the deliberator set a goal point that was a nowhere near the gap. Thus some gaps can be negotiated without much assistance from the deliberator, but for the tightest gaps the problem of sensing still remains. To make HYSTE perform at the peak of its abilities it must be able to sense gaps.

HYSTE is fully functional in the present configuration. It can thoroughly explore unknown areas. The scenarios shown above all display successful runs of HYSTE, and once localization is integrated HYSTE should be capable of exploring elaborate and difficult unknown areas. There are, of course, any number of improvements that could be made. A number of these are discussed in the conclusion and future work section (Section 8).

8 Conclusions and Future Work

8.1 HYSTE Successes

I believe that the system described in this paper embodies an agent architecture that is well suited to the task of navigating in an unknown space, although the paradigm could be expanded to other realms. First, the reactive layer represents a highly expandable and adaptable motor schema architecture. The schema system blends goals as subsumption never can, emulating the most successful agents in the known world, animals. A robot that can only do one task at a time, as is the case in a subsumption architecture, will sooner or later be faced with a situation in which the correct action is to attempt a combination of actions. When a robot avoids an obstacle while moving towards a goal, most subsumption architectures would invoke the avoid-obstacle behavior at the expense of an achieve-goal behavior. The obstacle will be successfully avoided, but the robot may nevertheless avoid the obstacle in a manner that is counter-productive to the main goal of the system, which is to achieve

a particular position. An agent using a schema architecture is never forced to forget the higher goals of the system. All goals are blended according to a shifting system of prioritization: for instance, the obstacle-avoidance schema should be more influential than the achieve-goal schema when an obstacle is about to be hit, but the subtle suggestions of the achieve-goal schema can help the robot avoid obstacles in manner that is ultimately advantageous to the greater goal of achieving a goal point. Schemata seem uniquely well suited to demands made by the task of real world navigation.

The schema-based system forms the core of HYSTE's reactive layer, but on its own it remains too static to deal with all the situations the world may contain. But the reactive system contains another level of abstraction: behaviors. Behaviors allow for real-time, adaptive parametrization, combining fixed schemata into cohesive groupings of practically infinite variety. Any fixed set of schemata is going to encounter situations that will lead to poor performance, but the intelligent switching of behaviors serves to cope with most difficult real-world scenarios. Secondly, behaviors create a paradigm for reactive/deliberative interaction. In this system, the interaction between the reactive and deliberative layers is intentionally kept to a minimum. The deliberative layer is slow and cumbersome, unsuited for the demands the real-world makes for fast action and reaction. But the deliberative layer, with its internal state, can perceive things that the reactive layer can't possibly notice given the reactive paradigm. However, instead of attempting to micro-manage the functioning of the reactive system, the deliberative layer can only set a goal point and change behavior. It can change the way the reactive system functions, but uninvolved in the actual motor decisions made by the reactive system. This leaves the reactive layer free to perform a low-level, animal-like reactive problem-solving strategy, trying all obvious strategies for getting around an obstacle. The reactive layer can move through gaps that the deliberative system can't even sense, and the deliberative layer can alter goal points and behaviors to allow the robot to pass through gaps that the reactive layer would be unable to negotiate on its own.

8.2 Shortcomings and Future Work

8.3 Basic Functional Changes

In the near future I intend to implement three modifications to improve the functioning of HYSTE. As discussed in Section 7.3, sensing gaps is extremely difficult. Currently, the evidence grid code was adapted from code written by Moravec [20]. The code is extremely thorough, but requires an obscene quantity of processing power. Currently, the evidence grid functions take up approximately 90% of the processor, and even then runs only about one time a second. Running the evidence grid this infrequently means that only one sensor set out of three or four is entered into the evidence grid. Running the evidence grid much more frequently could allow for much quicker exploration, as areas would resolve very quickly. Also, HYSTE currently makes little use of vision, but if that use is to be expanded, the processor must be freed up. Finally, there is no documentation for Moravec's code. During experimentation some of the parameters were altered, yielding very different looking evidence grids. It was impossible given time constraints to test enough to determine which parameters altered which aspects of the evidence grid, and what an optimal parameter configuration might look like. For all of these reasons, HYSTE needs new, efficient evidence grid code. Already HYSTE's performance is impeded by the nature of IR and sonar sensors, and it is important to make sure that the sensor information is combined in such a way to use it as effectively as possible.

Another small change that would improve the functionality of the system concerns the topological mapping performed by `mid`. Currently, the paths between connectivity nodes can be arbitrarily close to known obstacles as long as there is a line a single pixel wide that goes through unoccupied area. Thus the sequence obtained from manipulating of topological mapping can actually result in a poor path through known space. It would be helpful to alter the construction of the mapping to be much more like connectivity Voronoi diagrams constructed by Thrun et. al. [10]. These mappings attempt to construct paths that are as far away from obstacles as possible. The nature of the current topological mapping can make navigating through known space one of the most difficult tasks for HYSTE when it should be one of the easiest.

A final basic addition could improve the speed of the system, allowing HYSTE to explore an area faster and more efficiently. As the area being explored by HYSTE gets larger and larger, the time overhead for moving to any previously-visited pixel in the entire area becomes very high. Simply selecting the very best area over the entire evidence grid as the next goal point might explore new area, but it could be extremely time-consuming to move back and forth across the entire space. A better strategy for exploring area quickly might be to explore the closest area that appeared to have a viable frontier. This way, the distance that must be traveled through known space could be reduced. These simple changes could optimize the function of HYSTE considerably.

8.4 Learning and Adaptation

There is no learning in this system. All parameters were experimentally determined by hours of making small changes and observing the robot in different scenarios. The behaviors as formulated in this paper are attempts to group parameters in such a way as to achieve desired behavior. But I could not test limitlessly, and every behavior is a compromise. The behaviors are also finite, a few selections from a truly enormous parameter space involving all permutations of 50 parameters in the reactive layer alone. Many would say that learning systems are at their finest in foraging through large parameter spaces, trying countless possibilities until truly capable groupings are found. I agree with these claims and inevitable criticisms. The system is not a fully generalized one. Changing the average size of rooms will severely hurt performance. The behaviors listed here are at their best in tight spaces with few gaps; they are too slow and too thorough for moving through open spaces in which there may be only a few important tight places. A system that learned on a large number of different types of environments could likely do a better job of operating in a wider range of real-world domains.

Given all of these factors, adding learning to the system is a definite goal of future work. There are many reasons why there is no learning in the system as currently formulated. Genetic algorithms [13], a very effective way of finding reasonably good solutions in large parameter spaces, need large populations run for many generations. Neural networks, while they can be fabulously powerful, require extensive training and an explicit definition of success or failure for most standard learning techniques like backwards propagation [29]. Most researchers who attempt to implement learning on robots do so in simulation, but there are several problems associated with this. For one thing, there is no simulator currently available for the Magellan. Secondly, this system is designed for real world use. No simulator can fully capture the chaos that is real-world robotics, although some simulations do attempt to simulate real-world sensor noise [24]. Sensors go haywire, wheel encoders rapidly become

inaccurate, chair legs appear out of nowhere, and motors respond sluggishly. The parameters produced by learning in simulation can be useful as initial parameter values when beginning real world tests, but inevitably display their shortcomings when transferred to actual robots functioning in the non-simulated world. The final option is to try to train the robot in the real world. The Magellans are about 50 pounds, and can push a chair across a room at a half a meter a second. They are powered by two car batteries that give about 2 hours of life fully charged. A tether would have to consist of a very long 24V thick cord. In other words, these robots require constant supervision. Training them would require countless hours of shifting foam-board around, wasting time charging batteries, and dealing with inevitable software bugs: not a pretty picture. Thus while learning may offer a path to the creation of to truly adaptable robotic agents, adding learning to HYSTE is definitely a task for the future.

8.5 Better Sensing

I have discussed at length the shortcomings of sonar and IR sensors, but in all fairness they are inexpensive and are generally quite effective. They certainly should not be discarded, but perhaps forcing them to exceed their capabilities is not the best way to go about getting the most accurate picture of the world. One answer is to add more sensors with different capabilities. A single laser range finder, for instance, on the front of the robot, has millimeter precision and does not have the same poor performance on corners, although reflective surfaces could be a problem. The range finder range readings could simply be included in all of the sensor readings and would help considerably with door detection. Sonars have no problem with light reflective surfaces, and IRs have no problem with most kinds of textured surface. If all three of these sensors could be combined, doors and corners could be resolved quickly and reliably, drastically improving the performance of the system.

The Magellans have extremely nice pan-tilt high resolution color cameras, yet they were only used in HYSTE to detect the small green localization landmarks. Adding more vision processing to the system could improve the performance of HYSTE. For instance, algorithms exist for what is called ground-plane obstacle detection [16]. These algorithms utilize the fact that an obstacle is generally going to be a different color than a uniformly colored floor. Thus if the pixels that represent the floor can be removed from an image, the pixels that remain must be non-floor obstacles. Some algorithms, given a uniformly colored ground-plane, can give highly reliable depth information about obstacles, reporting how far the nearest obstacle is for obstacles that no range-finder type could find, like chairs and other thin objects. Chairs and other objects that are thin or have overhangs, like tables, are completely impossible for range-finding sensors to detect. Additionally, there are highly accurate door detectors that can parse the edges from doors and give accurate estimates of position and width [28]. The first and easiest expansion of HYSTE would be to utilize vision in two ways. The first would be to add a vision-based portion to the obstacle avoidance schema. A vision-based obstacle avoidance algorithm would report an estimated distance to the nearest obstacle, which would be factored into the obstacle avoidance schema. This could prevent collisions with chairs or other difficult to sense obstacles. The second addition would be to add a vision portion to the planning section. If a likely gap was sensed by vision in an area that had not yet been explored, the deliberative module could set that gap as a goal point [28]. The vision system could also be highly skilled at evaluation of a likely frontier that was directly in front of the robot. A region that seemed like a door when viewed on the evidence grid could be filled with a difficult to sense obstruction, or a door leading to unexplored

area could be obstructed by a small, easily movable obstacle. Range sensors would be unable to sense accurately in these situations, yet vision could be used to identify or reject a frontier.

8.6 Putting It all Together: Urban Search and Rescue

HYSTE was designed as a generalized exploration agent for exploring completely unknown areas. But there are likely applications for which the system could exist as a navigation backbone. One of these applications is an Urban Search and Rescue scenario [26]: the robot is dropped into a building of which it has little or no knowledge. It must explore as much area as possible, mapping the region and discovering victims. The scenario consists of a truly inhospitable world, filled with rubble, completely unknown, in which both speed and thoroughness are of the essence. For this application the system as it exists now could form the backbone of a Rescue agent, with several basic additions. First, a substantial vision module would need to be added, including the additions suggested above and a few more specialized features. The vision module would need several new capabilities, such as a victim-detector. The victim-detector might segment out flesh-tones or bright clothing colors from an image, or detect movement. Also, detecting obstructions that might be movable would be an important feature of the USR vision system. Venetian blinds, sheets, or movable rubble, which would look as solid as any wall to any range sensor, could be sensed by vision. When vision detected such a thing, it could work with the deliberator to set a goal point behind that obstacle, setting a very, very aggressive behavior for the reactive system. The behavior would be designed to bump the obstacle; if it was truly movable the bump sensors would not be depressed and the robot could move beyond the obstacle. An immobile obstacle would cause the bump sensors to be depressed, signalling that the vision module was incorrect about the object's movability.

The system is already designed to explore and map an unknown region, but a few final capabilities could be added. There might be a time constraint, such that the robot had 20 minutes to explore, and then must find its way back to the point of entrance. This could be implemented easily by a timer, which would follow topological links back to a starting point when the timer expired. Finally, multiple agents capable of exchanging state information could be used. Two or more robots could explore the space, exchanging information as they met, paying special attention to areas that neither had yet explored. The system is designed to work in harsh environments, in decidedly difficult arenas for normal robot navigation, although the Magellans are wheeled robots and require a flat floor surface for normal operation. As it is currently parametrized, HYSTE is designed to explore thoroughly, but not necessarily quickly. Part of the challenge of Urban Search and Rescue as an application for this system will be to parametrize HYSTE to combine speed with thoroughness.

A Schema Appendix

A.1 Sensor Processing

Using a combination of sensors can increase the reliability of the readings returned, but there are a variety of difficulties associated with combining sensor values. HYSTE sensor processing tends toward the conservative, in that the lowest of the two range sensor values is generally taken as the sensor reading for that pair, as the sensors are unreliable in predictable ways. Remember that a low sensor reading indicates that an obstacle is close. A sonar sensor that does not sense something within its range returns a value of 8000, meaning that it did not receive the echo from its ping. The sonars become unreliable at distances of less than approximately 25 centimeters, and may return a nonsense reading. Sonars will also return readings of 8000 if the ping bounces off the obstacle poorly and does not return to the firing sonar. Thus taking an 8000 sonar reading to mean that there is no obstacle in 3 meters in a particular direction may cause the robot to ram into unseen corners. The IR sensors are more accurate at short range but return a nonsense reading at values above approximately 28 centimeters, generally returning a random value between 400 and 500mm. IRs will generally detect corners and other strangely shaped obstacles that cause sonars to fail, but are only really useful at detecting obstacles in the immediate vicinity of the robot. The randomness of the IR reading when there is no obstacle in range means that a simple minimum can not be used to determine the lowest of the two sensor readings, as the IR reading might be nonsense. A sonar reading of 800 mm might be accurate, even though the IR sensor reports an obstacle at 450mm. Thus, there is an IR threshold parameter `ir_thresh` within the sensor processing portion of the navigation module that can be altered by the application dependent on the goals of the behavior. If an IR sensor reading is beneath this threshold and the IR sensor reading is less than the sonar reading, then the IR reading becomes the minimum reading for that pair. Otherwise, the sonar reading becomes the minimum.

It may be useful to adjust this threshold for different behaviors. A behavior which is not designed for operation in close proximity to walls may be well served by an IR threshold at around 18 to 20 centimeters, as sonar readings tend to be a bit more accurate in ranges above about 23 centimeters. A behavior designed for operation at close quarters might perform better with an IR threshold set to 25 centimeters, as IR sensors are better at detecting corners and other highly textured surfaces when an obstacle is in range. Generally, a robot using a behavior with a high IR threshold will act more conservatively at close range, whereas a robot using a behavior with a low `ir_thresh` value will rely more on unreliable sonars, navigating aggressively but somewhat blindly.

A.2 Obstacle Avoidance

Different behaviors may wish to cause the robot to avoid obstacles in drastically different manners in accordance with the goals of each behavior. Thus there are a number of parameters which can be altered to effect change in obstacle avoidance behavior. The first set of parameters has been dubbed `care` values. `Care` values specify the value under which a sonar reading should begin to cause a repulsive force. The range of the sonars is about 3 meters, but there are few applications in which it is desirable for agent to be shying away from obstacles 250 centimeters away. If the behavior is designed to function when the robot is moving through relatively unobstructed space, it might be desirable for the robot to start reacting to obstacles when they are still as far 150 centimeter, but for behaviors operating in tight spaces, in hallways or doors, the robot should ignore obstacles as close

as 60 centimeters. `Care` values allow behaviors to specify exactly what range should be reacted to. There is a `care` value for each of the four directions.

In a potential fields approach, obstacles exert a force along a distance gradient. This means that obstacles very close to the robot should exert a stronger repulsive force on the robot than obstacles farther away. In this implementation, there is a linear relationship between the distance of an object and the repulsive force it exerts. The IR minimum range, about 10 centimeters, and the `care` value for a particular direction form the range for the force; sensor readings below 10 centimeters will yield a force of 1, and readings above the `care` value will yield a force of 0. Sensor weightings for each direction are used to transform the sensor values into actual numbers representing the translation and rotation recommendations of the obstacle avoidance schema. These recommendations will eventually be summed with other schemata recommendations and passed to the motors. Every sensor pair that has a non-zero weighting in a particular direction has its reading passed to a function that scales the reading between 0 and 1. The scaled number is then multiplied by the direction-dependent weighting for that sensor pair. A sensor reading of 1 from a sensor pair that has a weighting of .3 will result in a scaled, weighted value of .3. Finally, the highest scaled, weighted value is selected for that particular direction. This system of sensor processing guarantees that readings from the most important sensors in a particular direction will be more important than readings from less important sensors, and will result in higher pushes; still, a less important sensor pair with a low reading will still result in a repulsive push if the more important sensors in a particular direction do not detect an obstacle. Sensors are unreliable, so it is important that less important sensors are not completely ignored, but giving all sensors in a particular direction equal importance can limit the spaces that a robot agent can explore.

The above weighting, scaling, and largest number selection have yielded a number d, between 0 and 1, but this number is not yet in units that make sense as motor commands. Another set of parameters influences this conversion: `max` values. The scaled value d is multiplied by the `max` value for that direction, yielding a number which could be passed to the `Mage` interface and result in a reasonable robot reaction. The `max` values represent the maximum obstacle-avoidance push in each direction, as they will always be multiplied by a value between 0 and 1. By this process, each direction recommends a number, scaled between 0 and `max`, as its recommendation for that direction. The final rotation recommendation for the obstacle avoidance scheme is the sum of the left and the right directional recommendations, and the final translation recommendation is the sum of the forward and the backward recommendations.

After obstacles avoidance recommendations have been computed, and the rest of the schemata make their recommendations the translation and rotation recommendations from all of the schemata are summed and passed to the motors as the commands for that time step. In this manner all the schemata, including obstacle avoidance, have their action values for the time step summed into the motor commands of the robot.

A.3 `Bump` Schema

It was decided to link bump sensor values with the sensor weightings discussed above because different behavior may want the robot to react to being bumped in a particular direction much as

it reacts to a low sensor reading in that direction. The main parameter in the `bump` schema is `bump_force`. `Bump_force` is the maximum translation and rotation recommendation that can be issued by the `bump` schema. `Bump_force` is scaled by two bump vectors, each of which contains an entry from 0 to 1 for each of the 16 bump sensors. The two vectors correspond to the translation and the rotation scaling factors for each bump sensor. Thus if a given bump sensor b is depressed, the `bump` schema recommends a translation of `bump_force` $*$ `bump_trans_vector`[b] and a rotation of `bump_force` $*$ `bump_rotation_vector`[b]. When a given bump sensor is depressed, it will report that it has been pressed for several cycles, which generally makes the robot move away from the obstacle if the parameters are set appropriately. Different behaviors may specify different `bump_force` values, however, depending on the preferred bumping behavior of that robot. A behavior designed to cause the robot to actually push obstacles may need to set this value quite low. This type of behavior may even set a negative `bump_force` value, which would push the robot into obstacles when a bump sensor was depressed.

A.4 `Goal` Schema

The parameters governing the `goal` schema can be difficult to understand. The translation recommendation, which will remain constant until altered by a new behavior, can be adjusted. This parameter largely governs the speed at which the robot moves through the world. The parameters associated with the rotation recommendation are a bit more complicated. The angle of the robot to the goal point is computed in thousandths of radians. There are 6.28 radians in 360°, and thousandths of radians are being used for the unit, so headings are in a range between -3141 and 3141, such that 0 means that the robot is pointed directly at the goal. Three parameters control the conversion of this angle value into the rotation recommendation for the `goal` schema. The computed angle to the goal a, a range parameter `max_angle`, and a range parameter `min_angle` are passed to a function which linearly scales a with respect to `max_angle` and `min_angle`. This function returns a scaled value s, where $s = 1$ if $|a| \geq$ `max_angle`, $s = 0$ if $|a| \leq$ `min_angle`, and $s = (a -$ `min_angle`$)/($`max_angle` $-$ `min_angle`$)$ if `min_angle` $\leq a \leq$`max_angle`. The scaled value s is then multiplied by another parameter `max_rotate_val`, the magnitude of the maximum rotation push that can be recommended by the `goal` schema. The product of $s *$ `max_rotate_val` should yield a value between 0 and `max_rotate_val`. This value is then given the correct sign, positive if $a \geq 0$ and negative if $a < 0$. This value is reported as the rotation recommendation for the `goal` schema.

The workings of these three parameters can be a bit confusing. Adjusting `max_angle` and `min_angle` allow a range of types of goal pushes. Some behaviors might be designed to move the robot very directly towards a goal without deviating from a strict goal heading. This kind of goal behavior could be created by achieving a maximum rotation push very quickly by setting a low value for `max_angle`, and a low or zero `min_angle`, which would push to correct any deviation from a zero heading very quickly. Another behavior might be designed to give the robot more latitude, not forcing it to move directly towards the goal in a dedicated fashion. This flavor of `goal` schema could be created by setting high values for both `max_angle` and `min_angle`, so that rotation corrections would not be made at all until the current heading was substantially different than the goal heading, and the maximum recommendation would not be achieved until the current heading was extremely far from the goal heading. The `max_rotate_val` parameter can be used to make the goal schema more or less influential in relation to other schema. A high `max_rotate_val` can cause the `goal` schema to dwarf the influence of other schemata, and a low value can lessen its importance.

A.5 `Stuck` Schema

The first set of parameters governing the function of the `stuck` schema designates which final translation and rotation values constitute a local minimum. If the total summed translation and rotation values from the previous cycle are less than `stuck_trans` and `stuck_rotate` respectively, then a counter is incremented. If the translation value from the last cycle is greater than `stuck_trans`, or the rotation value is greater than `stuck_rotate`, then the counter is reset to 0. Thus initialization of stuck behavior requires a number of consecutive low-translation, low-rotation cycles. The number of consecutive low-translation, low-rotation cycles necessary to initiate the response of the `stuck` schema is set by another parameter, `stuck_consec`. If the counter ever becomes greater than `stuck_consec`, then the `stuck` scheme begins recommending rotation values designed to move the robot out of the local minimum.

Once the `stuck` schema has decided that the robot is at a local minimum, another group of parameters governs the nature of the rotation recommendations to move the robot away from the minimum. The intensity of the rotational push is specified by `stuck_intensity` and the direction to rotate is set by `stuck_direction`. When the `stuck` schema is triggered, it recommends a sharp turn for a certain number of cycles. The number of cycles is determined in two ways. The first method is simply to recommend a turn for a number of cycles specified by `stuck_duration` and to assume that the robot is away from a local minimum after that number of cycles. Suppose, however, that the robot is at a corner and is stuck at a local minimum. The `stuck` schema recommends a turn to the right for ten cycles, directly into the other side of the corner. The ten cycles pass, the `stuck` recommendation ends, and the robot is briefly away from the local minimum. The repulsive force of the other wall then comes into play and pushes the robot back towards the other side of the corner, straight back into the local minimum it just exited. To avoid this type of infinite cycle, the `stuck` schema can be parametrized to continue the turn recommendation until there is no obstacle in front of the robot closer than a certain distance `obs_thresh`. This ensures that the robot will be able to move forward slightly when the `stuck` schema ends the turn recommendation, which generally moves the robot permanently away from a local minimum.

Parameters for the `stuck` schema must be set very carefully, as there are situations in which turning away from what seems to be a minimum can be counterproductive to attaining a goal point. Specifically, the robot may appear to be in a local minimum when in difficult positions, such as trying to go through a door it is facing, which is a surprisingly difficult situation, discussed at length in Section 5.2. The `stuck` schema may deem that the robot is at a local minimum when in fact the robot simply needs to make small adjustments to get through the door. In this fashion, the `stuck` schema may recommend a turn that keeps the robot from moving through a door that would lead to a goal point. This makes setting the parameters for the `stuck` schema very difficult, as ideally the robot should move quickly from real local minima, but not turn away from situations in which allowing a little more time could result in successfully navigating a tight place. The perfect set of parameters for achieving these two goals is quite difficult to find. Finally, for the `stuck` schema to be effective at moving the robot away from local minima, it requires that the `stuck_intensity` be set high relative to the `goal` schema rotation maximum and the left-right obstacle avoidance `max` values. Otherwise, the robot may remain in a local minimum even after the `stuck` schema begins trying to turn the robot. But even with a relatively high `stuck_intensity`, there can be difficulties. Suppose the robot is facing a wall that blocks its progress to a goal point and there is another wall to its right. After a

number of consecutive low movement cycles, the `stuck` schema begins turning the robot to the right with an intensity of 550 units. But the `goal` schema pushes the rotation back left 300 units, and the wall to the right results in a leftward obstacle avoidance push of 250. Even with the `stuck` schema recommending strong pushes, the robot can remain in a local minimum.

A.6 `Fluct` Schema

Many local minima can be effectively avoided by using the `stuck` schema as described above. But there are situations in which the robot can become stuck despite the efforts of the `stuck` schema. Thus there is another minima avoidance scheme, dubbed the fluctuation scheme, or `fluct`. Some approaches to motor schema use a noise schema to avoid local minima. A random value is added to the translation or the rotation values in the hope that this will provide enough variation to move the robot from local minima. I experimented with this approach, and decided that it was not that effective given my formulation of motor schema. The weakness of the noise schema appears to involve real world properties of the motors. When the motors receive a command, they must accelerate or decelerate to achieve the desired speed. This acceleration takes time, and certainly more time than the single cycle of the four or so cycles per second in which the robot evaluates its environment. If a random value of 40 is picked at one cycle, and the next cycle yields a random value of -40, then the motors effectively will not move at all. For a noise schema to effectively dislodge the robot, the pushes would have to be of such a great intensity that they might cause the robot to run into a wall or do something similarly misguided.

After experimenting with various formulations of noise schema, I decided to try to implement a variant of the basic noise schema. Instead of picking a random value every cycle, I decided to try to make the noise recommendation vary smoothly with time, to avoid the jerkiness associated with a random noise push. The result was the `fluct` schema, which uses three parameters to compute a noise value. The first parameter, `fluct_period`, sets the periodicity of the standard `sin` function that is used to give a smoothly varying noise recommendation. In addition, there are two range values, one for each translation and rotation, `fluct_trans` and `fluct_rotate`. The rotation recommendation is computed as follows. Suppose that `fluct_period` is set to 10 (seconds) and `fluct_rotate` is set to 75. If n is the number of seconds since initialization, the fluctuation schema will vary smoothly from a recommendation of -75 when $n\%10 = 0$, to recommending 75 when $n\%10 = 5$, and back to -75 when $n\%10$ is again 0. The translation recommendation is computed in exactly the same fashion using `fluct_trans`. With this schema functioning, the robot looks as if it is doing a "waggle," moving slowly from side to side.

As it turns out, the combination of the `fluct` schema with the `stuck` schema means that the robot rarely gets stuck in a local minimum for long. With just the FLUCT running, there are situations that leave the robot never quite moving away from local minimum. But with both the `stuck` and `fluct` schemata running, minima are generally completely avoided. Additionally, the "waggling" has additionally helpful characteristics for the attaining of goal points, as discussed in Section 5.2.4.

CRUISE	FAST_SAFE	SLOW_SAFE
sensor_model[FRONT][15] = .8	sensor_model[FRONT][15] = .75	sensor_model[FRONT][15] = .5
sensor_model[FRONT][1] = .8	sensor_model[FRONT][1] = .75	sensor_model[FRONT][1] = .5
IR_trust = 225	IR_trust = 240	IR_trust = 220
care[FRONT] = 1000	care[FRONT] = 800	care[FRONT] = 800
care[LEFT] = 800	care[LEFT] = 800	care[LEFT] = 800
care[BACK] = 800	care[BACK] = 800	care[RIGHT] = 800
care[RIGHT] = 800	care[RIGHT] - 800	care[BACK] = 800
max[FRONT] = 350	max[FRONT] = 400	max[FRONT] = 220
max[LEFT] = 350	max[LEFT] = 450	max[LEFT] = 500
max[BACK] = 200	max[BACK] = 200	max[BACK] = 100
max[RIGHT] = 350	max[RIGHT] = 4500	max[RIGHT] = 500
bump_force = 500	bump_force = 500	bump_force = 500
max_rotate = 450	max_rotate = 360	max_rotate = 400
max_angle = 1000	max_rotate = 1300	max_angle = 1100
min_angle = 40	min_angle = 200	min_angle = 100
push_trans = 130	push_trans = 120	push_trans = 100
goal_follow = 0	goal_follow = 1	goal_follow = 0
stuck_trans = 50	stuck_trans = 90	stuck_trans = 60
stuck_trans = 90	stuck_rotate = 200	stuck_rotate = 80
stuck_distance = 400	stuck_distance = 450	stuck_distance = 400
stuck_intensity = 450	stuck_intensity = 450	stuck_intensity = 450
stuck_direction = 0	stuck_direction = 0	stuck_direction = 0
fluct_rotate_period = 8	fluct_rotate_period = 6	fluct_rotate_period = 4
fluct_rotate = 30	fluct_rotate = 40	fluct_rotate = 40

Figure 27: Behavior parameter definitions for behaviors used in HYSTE

AGG	CRUISE_AGG	VERY_AGG
sensor_model[FRONT][15] = .4	sensor_model[FRONT][15] = .4	sensor_model[FRONT][15] = .3
sensor_model[FRONT][1] = .4	sensor_model[FRONT][1] = .4	sensor_model[FRONT][1] = .3
care[FRONT] = 700	care[FRONT] = 500	care[FRONT] = 700
care[LEFT] = 700	care[LEFT] = 500	care[LEFT] = 700
care[BACK] = 700	care[BACK] = 500	care[RIGHT] = 700
care[RIGHT] = 700	care[RIGHT] = 500	care[BACK] = 500
max[FRONT] = 250	max[FRONT] = 400	max[FRONT] = 200
max[LEFT] = 500	max[LEFT] = 450	max[LEFT] = 500
max[BACK] = 200	max[BACK] = 200	max[BACK] = 100
max[RIGHT] = 500	max[RIGHT] = 450	max[RIGHT] = 500
bump_force = 500	bump_force = 500	bump_force = 500
max_rotate = 500	max_rotate = 500	max_rotate = 450
max_angle = 1000	max_angle = 800	max_angle = 1000
min_angle = 40	min_angle = 40	min_angle = 70
push_trans = 250	push_trans = 140	push_trans = 140
goal_follow = 0	goal_follow = 0	goal_follow = 0
stuck_trans = 60	stuck_trans = 30	stuck_trans = 60
stuck_rotate = 80	stuck_rotate = 70	stuck_rotate = 80
stuck_distance = 400	stuck_distance = 450	stuck_distance = 350
stuck_intensity = 400	stuck_intensity = 400	stuck_intensity = 500
stuck_direction = 0	stuck_direction = 0	stuck_direction = 0
fluct_rotate_period = 8	fluct_rotate_period = 6	fluct_rotate_period = 4
fluct_rotate = 30	fluct_rotate = 40	fluct_rotate = 40
fluct_trans = 50	fluck_trans = 0	fluct_trans = 60

Figure 28: Behavior parameter definitions for the behaviors used in HYSTE.

B Behaviors

B.0.1 CRUISE

This behavior is intended to be able to deal with most situations caused by using the topological map to navigate through known space. The sequence of goal points generated by manipulations of the topological map may lead the robot through very small gaps however. In this case, the robot, using CRUISE, may fail to reach the goal point, and CRUISE_AGG, a more aggressive form of CRUISE, will generally be called to cause the robot to move through the gap. Thus CRUISE is parametrized to move the robot efficiently and quickly to goal points, navigating between all but the tightest gaps. Please see the schema table (Figure 28) for the exact definition of CRUISE behavior. Care values are low in CRUISE, except the front care value. CRUISE should not cause the robot to react to obstacles to either side until those obstacles are fairly close, but early reaction to front obstacles can avoid some local minima. The max values are relatively low to the front and relatively high to the sides. The robot with CRUISE behavior should to be move near to walls to achieve goal points, hence the low front max value. The combination of low side care values with high side max values means that obstacles near the robot will cause substantial pushes, and all other obstacles will cause no push at all. If a side obstacle is within the care range, the robot should move substantially away from it, but otherwise it should be ignored. CRUISE should not cause the robot to veer from the goal path unless it is completely necessary that it alter its path.

Four other schemata besides obstacle avoidance are functioning in this behavior. The bump schema is on, with a high bump force, just in case something unexpected occurs. As CRUISE is for moving the robot to goal points quickly, the goal schema is of course on. Again, the goal schema for CRUISE looks very different than for the exploration behaviors, as it is meant purely to be a goal point achiever. The max_rotate value is a little higher than in other behaviors, as the robot should move directly towards the goal. The min_angle value is set slightly above 0, to allow the fluct schema to function without having its recommendation immediately nullified by a goal push. The most drastic difference is that max_angle is set to half of the value to which it is set in the other behaviors. The behaviors designed more for exploration should allow the robot to turn away from the goal point to some extent to allow for substantial exploration if the path to the goal is blocked. For CRUISE however the robot should not be allowed that leniency, as the robot should never need to turn very far away from the goal in its normal course of operation. As noted above, the push_trans value is relatively high, as the robot should achieve the goal point as quickly as possible.

For local-minima avoidance, the fluct schema is also on, but in this case acts less as a local minima avoidance schema than a "waggler." Waggling is a word taken from the biology of bees and other insects, which perform a "waggle" dance, shaking their behinds to give directions to a food source. In this case, waggle is meant to consist of a slight rotational shift that alternates sides in quick succession. The robot will rotate slightly to the left, and then back to the right, and then to the left again, going just a bit past the center each time. For the robot, this is just a way of getting better sensor readings off from a hard-to-sense obstacle. Sonars can bounce poorly off of angles or other features of an environment. By waggling, the front sensors can bounce pings off an obstacle at a number of angles, and are more likely to get good readings. Thus the fluct schema in this behavior causes a slight waggle to ensure that a forward obstacle is sensed quickly and reliably. Finally, the stuck schema is

functioning, but with very low stuck_trans and stuck_rotate values, as the robot should turn away from the goal path only if there is no hope of escaping from a local minima except by turning away.

B.0.2 SLOW_SAFE

First, the front right and left sonar weightings are reduced slightly, to allow the robot to move through doors slightly better than in FAST_SAFE mode. The back side sonar weightings are likewise slightly reduced for the following reason: when the robot is following a wall to the right and the goal point lies beyond the wall, then any hole will cause the force pushing the robot away from the wall to disappear, and the robot will turn towards the gap. If all four of the side sonars have an equal weighting, then the force caused by the wall will not disappear until after the last of these sensors is past the wall, causing the robot to turn into the gap less quickly. If the back side sonars were weighted less than the front side sonars, the robot would turn into the gap earlier, giving it a better chance of successfully moving through the gap. The trade-off is that when passing a thin obstacle to the side, if that obstacle does not push away as much at the back side sonar positions, the robot may turn into the obstacle and bump it while trying to achieve a goal point. For the door-finding behaviors, however, the trade-off is worth it.

The SLOW_SAFE behavior sets several other parameters differently than FAST_SAFE does. The care values are all turned down slightly, as the robot should not be distracted by distant obstacles. The front max value is also reduced, as this behavior should be able to cause the robot to move near walls in front. The forward goal push has been reduced, to slow the robot down (hence, SLOW_SAFE. SLOW_SAFE is meant to be a more thorough explorer than FAST_SAFE, and as such the stuck parameters look quite a bit different. Both stuck_trans and stuck_rotate are lowered, and stuck_threshold raised, requiring more consecutive cycles of lower final motor recommendations to initiate stuck behavior. Also, stuck_distance is lower than in FAST_SAFE, meaning that the robot has to be very close to an object to continue turning past turn_period cycles.

Finally, the bump schema remains unchanged, and the fluct schema just has slightly lowered parameters.

B.0.3 CRUISE_AGG

During testing of the planning module, it became clear that CRUISE as specified above was not sufficient to navigate along the paths set by the topological landmarks. The landmarks are all connected by unobstructed lines, but these lines can be quite close to walls, which will cause the CRUISE behavior to fail where a slightly different behavior can succeed. CRUISE_AGG is different in that it is only chosen by the deliberative layer when the deliberative layer is very sure that a path exists to a goal point. Thus CRUISE_AGG exists not to cause the robot to explore, but to aggressively achieve a goal point even if the path to that point takes the robot through small gaps and very near obstacles. CRUISE_AGG is parametrized to cause the robot to ignore obstructions until the last possible moment and then make only slight deviations from the goal path.

The obstacle avoidance parameters for CRUISE_AGG look quite different than any those of any other schema. The care values are especially low, as the robot in CRUISE_AGG mode should not respond to obstacle influences unless it absolutely must. The max values are also set fairly low, as obstacles cause

a repulsive force unless they are extremely close to the robot. Similarly, the weightings for all but the most important sensors are drastically reduced. The `bump` schema may be particularly important for this behavior, and is definitely functioning. The `goal` schema is parametrized much like `CRUISE`'s, with no wall-following, a high `max_rotate_val`, and low `max_` and `min_angles`, to create a dedicated goal-seeker. The translational push is toned down to some extent, however, as this behavior is intended to operate in close proximity to obstacles. The `fluct` schema is left on with a slight rotational value, to allow for a little waggling. Finally, the `stuck` schema is parametrized so as to virtually disable it, as this behavior should cause the robot to barrel on through rather than turn away from the goal.

C Figures

List of Figures

D Bibliography

References

[1] M.A. Arbib and D. House. Depth and detours: An essay on visually guided behavior. In M. Arbib and A. Hanson, editors, *Vision, Brain, and Cooperative Computation*, pages 129–63. MIT Press, Cambridge, MA, 1987.

[2] R. C. Arkin. *Behavior Based Robotics*. MIT Press, Cambridge, MA, 1998.

[3] R.C. Arkin. Motor schema based navigation for a mobile robot: An approach to programming by behavior. In *Proceedings of the IEEE Conference on Robotics and Automation*, pages 264–71, 1987.

[4] R.C. Arkin. Motor schema-based mobile robot navigation. *International Journal of Robotics Research*, 8(4):92–112, 1989.

[5] M. Betke and K. Gurvits. Mobile robot localization using landmarks. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 2, pages 135–142, 1994.

[6] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, 1986.

[7] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[8] G. Dudek and M. Jenkin. *Computational Principals of Mobile Robotics*. Cambridge University Press, Cambridge, U.K., 2000.

[9] G. Dudek, M. Jenkin, E. Milios, and D. Wilkes. Robotic exploration as graph construction. In *IEEE Trans. Robotics and Automation*, 1991.

[10] S. Thrun et. al. Map learning and high speed navigation in RHINO. In D. Kortenkamp, R. P. Bonnasso, and R. Murphy, editors, *Artificial Intelligence and Mobile Robots*. MIT/AAAI Press, 1997. RR n3552 26 Laugier, Fraichard, Garnier, Paromtchik and Scheuer, 1997.

[11] N. Fairfield. Simple landmark localization on a three-layer mobile robot architecture. 2001.

[12] E. Gat. On three-layer architectures. In D. Kortenkamp, R. P. Bonnasso, and R. Murphy, editors, *Artificial Intelligence and Mobile Robots*. MIT/AAAI Press, 1997. RR n3552 26 Laugier, Fraichard, Garnier, Paromtchik and Scheuer, 1997.

[13] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machin Learning*. Addison-Wesley, Reading, MA, 1989.

[14] J. Gutmann, W. Burgard, D. Fox, and K. Konolige. An experimental comparison of localization methods. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'98)*, 1998.

[15] R. Haralick and L. Shapiro. *Computer and Robot Vision*. Addison-Wesley, Reading, MA, 1992.

[16] I. Horswill. Visual collision avoidance by segmentation. In *Proceedings of the IEE/RJS Internation Conference on Intelligent Robots and Systems*, Munich, Germany, September 1994.

[17] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *The International Journal of Robotics Research*, 5(1), 1986.

[18] B. Krogh. A generalized potential field approach to obstacle avoidance control. Technical Report SME-RI-MS84-484, Society of Manufacturing Engineers, 1984.

[19] C. Madsen and C. Andersen. Optimal landmark selection for triangulation of robot position. *J. Robotics and Autonomous Systems*, 13(4):277–292, 1998.

[20] M. Martin and H. Moravec. Robot evidence grids. Technical Report CMU-RI-TR-96-06, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, March 1996.

[21] M.J. Mataric. Environmental learning using a distributed representation. In *IEEE Int. Conf. Robotics and Automation*, volume 1, pages 402–406, 1990.

[22] B. Maxwell and L. Meeden et al. REAPER: A reflexive architecture for perceptive agents. *AI Magazine*, 2001.

[23] D. McFarland and T Bosser. *Intelligent Behavior in Animals and Robots*. MIT Press, Cambridge, MA, 1993.

[24] L. Meeden. Improving the gap between robot simulations and reality using improved models of sensor noise. In J.R. Koza et. al., editor, *Proceedings of the Third Annual Conference on Genetic Programming*, pages 824–831, San Francisco, CA, 1998. Morgan-Kaufman Publishers.

[25] D. Scharstein and A. Briggs. Real-time recognition of self-similar landmarks. Technical report, Middlebury College, 1999.

[26] A. Schultz. The 2000 AAAI mobile robot competition and exhibition. *AI Magazine*, 2001.

[27] G. Welch and G. Bishop. An introduction to the Kalman filter. Technical Report TR 95-041, University of North Carolina at Chapel Hill, Chapel Hill, NC, 1999.

[28] C. Wellington, R. Bock, and B.A. Maxwell. Finding door locations in an image. In *Intelligent Robot and Computer Vision XVII: Algorithms, Techniques, and Active Vision*. SPIE, September 1999.

[29] P. Werbos. Backpropogation: Basics and new developments. In M. Arbib, editor, *The Handbook of Brain Theory and Neural Networks*, pages 134–39. MIT Press, Cambridge, MA, 1995.