INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Hierarchical robot path planning using a distributed blackboard

Ciscon, Lawrence A., M.S.
Rice University, 1990
RICE UNIVERSITY

HIERARCHICAL ROBOT PATH PLANNING USING A DISTRIBUTED BLACKBOARD

by

LAWRENCE A. CISCON

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE MASTERS OF SCIENCE

APPROVED, THESIS COMMITTEE

Rui J. P. de Figueiredo
Dr. Rui J.P. deFigueiredo (Chair)
(Professor of Electrical and Computer Engineering and Mathematical Sciences)

D. H. Johnson
Dr. D.H. Johnson
(Professor of Electrical and Computer Engineering)

Ian Walker
Dr. Ian Walker
(Associate Professor of Electrical and Computer Engineering)

Houston, Texas
April, 1990
HIERARCHICAL ROBOT PATH PLANNING USING A DISTRIBUTED BLACKBOARD

by

LAWRENCE A. CISCON

Abstract

Hierarchical path planning techniques for Intelligent Autonomous Mobile Robots (IAMRs) attempt to subdivide the overall task of path planning into several layers. This subdivision is made more difficult by the physical robot's computational architecture. The most important problem with classic hierarchies is the its inability to cope with a diversity of environments and obstacles. In this research, we present a hierarchical path planning paradigm which uses a distributed blackboard architecture as its foundation. We subdivide the path planning problem into three levels: a global path planner, a local planner, and a set of constraint analyzers. By creating a uniform distributed blackboard that spans many processors in a network, we have formed the basis of a uniform operating environment for experimenting with layered path planning, as well as other mobile robot issues. And by dividing the planning into three layers instead of two, the system is able to handle much more complicated and diverse environments.
Acknowledgements

I would like to thank the members of my committee for their help in both the research involved in this thesis as well as their numerous comments and corrections. I would additionally like to thank Dr. Wise for the numerous comments, suggestions, improvements, and especially ideas, that he contributed to this work. It is certain that this research would not have progressed nearly as far without his thoughts.

Thanks are also due to those people who made possible my financial support through a NASA graduate student fellowship. And especially important was the contributions of Texas Instruments and Alex Meng of their knowledge, algorithms, and hardware that made the implementation of this research possible.
Table of Contents

1. Introduction .............................................................................................................. 1

2. Background ............................................................................................................. 4
   2.1 Autonomous Mobile Robot Computer Architectures .................. 4
   2.2 Basic Path Planning Strategies ......................................................... 11
   2.3 Hierarchical Path Planning Techniques ................................. 22
   2.4 Blackboard Systems ................................................................. 29
   2.5 Distributed Blackboards ........................................................... 35

3. Hierarchical Path Planning Architecture ......................................................... 42
   3.1 Subdivision of the Problem Space .............................................. 42
   3.2 Details of the Global Planner .................................................... 55
   3.3 Details of the Constraint Analyzer .......................................... 60
   3.4 Details of the Local Planner ....................................................... 65

4. Distributed Blackboard Architecture ................................................................. 71
   4.1 Design Decisions ........................................................................... 71
   4.2 Program Interface and Organization ......................................... 78
   4.3 Implementation of Transparency ............................................. 82
   4.4 Assumptions and Simplifications in our Design .................... 86

5. Advanced Path Planning Using Blackboards ..................................................... 90
   5.1 Fusion of path planning and the blackboard ......................... 90

6. Experimental Results ......................................................................................... 98
   6.1 Performance Monitoring and Debugging Issues ..................... 98
   6.2 Demonstration of Complete Hierarchy .................................... 99
   6.3 Analysis of Overhead and Throughput of Blackboard ............. 101
   6.4 Response Time of Planners ....................................................... 105
   6.5 Interaction of Blackboard and Planners ................................. 106

7. Conclusions ......................................................................................................... 108
8. References .......................................................................................................... 111
9. Appendix A. Implementation Details .............................................................. 118
# List of Figures

1. Block Diagram of Hermies .......................................................... 5
2. Block Diagram of CMU NavLab .................................................. 6
3. Picture of Rice-obot I .............................................................. 8
4. Block Diagram of Rice-obot I .................................................. 8
5. Robot in Normal Space and Configuration Space .............................. 12
6. Voronoi Graph and Generated Path ........................................... 14
7. Potential Field of Obstacle ..................................................... 17
8. Quadtree Representation and Encoding ....................................... 18
9. Quadtree Generated Path ....................................................... 19
10. LIDAR Scanner on Robot ........................................................ 20
11. Classic Planning Hierarchy ..................................................... 23
12. CMU NavLab System Architecture .......................................... 25
13. Brooks' Layering Technique ................................................... 27
14. Blackboard and Knowledge Sources ......................................... 31
15. Hearsay-II Architecture ........................................................ 33
16. Distributed Vehicle Monitoring Testbed Blackboard Node .............. 37
17. Transaction State Diagram ..................................................... 40
18. Floorplan of Model Environment ............................................... 44
19. Floorplan of Office .............................................................. 45
20. Three-level Hierarchy .......................................................... 53
21. Retraction Planning ............................................................. 58
22. Path Generation by Global Planner .......................................... 59
23. Encoding of Smooth Trajectories ........................................... 62
24. Detection of Collision with Overhead Obstacle ............................ 63
25. Control of Multiple Constraint Analyzers .................................. 64
26. Sensor Arrangement on Rice-obot ............................................ 68
27. Large Obstacle Detection ..................................................... 68
28. Small Obstacle Detection .......................................................... 69
29. Communications Level of Blackboard ...................................... 82
30. Registering and Distributing Objects ...................................... 86
31. Object Dependencies ......................................................... 94
32. Test Environment ............................................................. 99
33. Motion Correction for Obstacle .............................................. 100
Chapter 1

Introduction

Path planning for an Intelligent Autonomous Mobile Robot (IAMR) in an unstructured (i.e. real-world) environment is extremely complicated. Finding a single all-encompassing path generation algorithm which can establish both long-range goals and make effective use of multiple advanced sensors is difficult. A different approach than the all-in-one planner is being studied for several advanced IAMRs using a hierarchical system; these IAMRs include Carnegie Mellon's NavLab, the Autonomous Land Vehicle (ALV) and the EVA Retriever. A hierarchical path planner subdivides the path planning problem into a series of layers; each layer is assigned a particular subspace of the work environment. There is a path planner associated with each of these layers. Each of the planners may have its own representation of the workspace, and each of the planners has different constraints on the accuracy and speed with which it plans a path. Subdividing the path planning has several advantages, including the ability to handle more diverse environments and improved computational efficiency.

The most notable examples of hierarchical path planners are the systems in the CMU Navlab[40] and the Autonomous Land Vehicle (ALV)[44]. Both of these systems subdivide the robot planning into four layers: the Mission Planner, the Navigator, the Pilot, and the Controller. The Mission Planner is responsible for dividing the mission task (which is formulated by the operator) into a series of global motion goals. The Navigator receives these goals and is responsible for finding a path to each of these
goals. Given this global path, the Pilot attempts to navigate it using information from the sensors to avoid collisions with unexpected obstacles. The Controller uses encoders and other sensors on the base of the robot to accurately follow the path passed to it by the Pilot. This general hierarchy is used by many researchers as a basis for their planners, and we decided that it was a good place to begin our research. The main modification we have made to this hierarchy is adding an additional planning layer between the global and local planners. This layer consists of a set of constraint analyzers which analyze the global path for particular constraints, such as overhead objects, or rotations required to fit through tight spaces. This additional layer provides two important capabilities to the hierarchy: first, it provides a means by which domain-specific information about obstacles and the robot can be used to verify the global path before it is passed to the local planner. This lessens the possibility of the robot following paths that are impassible. Secondly, it provides a means by which small coarse corrections can be made to avoid obstacles or to maneuver without invoking the global planner. This improves the speed of the system.

In the CMU planner a pseudo-blackboard architecture for interconnecting the various pieces of the hierarchy is used; this system provides a means of sharing data between processes, timestamps for coordinating data taken from different sensors, and geometrical reasoning routines. We felt that a significant improvement in flexibility and modularity could be achieved in our system by creating a similar system for data transfer. However, we took a different approach to the blackboard by drawing upon ideas from distributed operating systems and databases to create a high-throughput, uniform blackboard interface.

In developing the hierarchical system, we touched upon a wide range of related sub-topics, including distributed processing, databases, blackboards, and algorithmic path planning techniques. There is a broad range of background information covered in these
topics. We began our research by studying IAMR computer architectures to determine the kinds of computers and communications the hierarchy would have available. Then we analyzed the strengths and weaknesses of some algorithmic path planning techniques, as well as some hierarchical path planners that other advanced IAMRs have used. Finally, we studied blackboards and distributed blackboard theories to form the underlying communications. We will present an overview of this background information in chapter two, including explanations of the shortcomings of existing systems.

Having introduced the issues and techniques, in chapter three we will outline the hierarchy we designed to overcome the shortcomings of other systems. We will also introduce the path planners we implemented to experimentally test each level of the hierarchy. Using this hierarchical model, we will describe the distributed blackboard architecture we formed to handle communications and process invocation. And then in chapter five we will show how the hierarchy is implemented in this blackboard environment. Finally, we will show experimental results of the overhead and response time of each part of the hierarchy in chapter six. There are two important results that our experiments immediately verified. First, the planning hierarchy successfully navigated in an obstacle-filled environment that contained multiple unexpected obstacles and other constraints on motion; these constraints would have made it difficult or even impossible for other path planning hierarchies to successfully navigate to the goal position. And second, we demonstrated that the overhead associated with the blackboard system was small in comparison to the computation time of the various pieces of the hierarchy. The overall throughput of the blackboard is large enough to handle major expansions of the hierarchy in the future.
Chapter 2

2.1. Autonomous Mobile Robot Computer Architectures

Many different computer systems are used in robots, differing in the degree of parallelism, the number of processors, the level of sophistication of the processors, and the interconnection systems. These factors affect the design of the path planning hierarchy by suggesting efficient distributions of tasks and by limiting the throughput we can expect to achieve. Furthermore, the kinds of interconnections determine how difficult it is to pass information from process to process. And the degree of sophistication of the software support (i.e. operating systems, debuggers, etc.) will constrain our choices in languages and debugging. We will look at three mobile robot architectures: the Hermies IIB robot of Oak Ridge National Laboratories, the Carnegie Mellon NavLab and the Rice University Rice-obot I. The Rice-obot is especially important to us because it is the robot on which we implemented our experimental system.

2.1.1. Example 1: Hermies-IIB Oak Ridge National Laboratory

Oak Ridge National Laboratory has been working on its Hermies series of mobile robots for several years, the Hermies-IIB being one of the latest incarnations. This robot was developed for researching navigation in semi-structured environments, such as a
nuclear plant or factories, and for researching simple manipulation of valves and controls[21]. The robot has three main computer processors on-board. The most powerful processor is an 4-node hypercube parallel processor. This system resides in an IBM AT-compatible host which acts as its gateway to external events. All of the control hardware resides on a VME cardcage with a 68000-based processor for driving motors and controlling sensors. A block diagram of the system looks like this:

![Block Diagram of Hermies](image)

**Figure 1. Block Diagram of Hermies**

The main operating system used on the AT is OS/9 which provides rudimentary concurrency control. The hypercube includes its own control program which communicates via subroutines with the AT host. The 68000-based computer is programmed in assembly language.

The custom engineered features of Hermies are indicative of most of the existing autonomous mobile robots. The VME to AT-bus connection, along with the access delays between the host computer and the hypercube, limit the throughput from the sensors to the hypercube to about .25 Mbps. This is equivalent to about one frame of
video per second. Hermies does not have a unified control system. Instead, the path planner, task planner, vision analyzer, etc. are individually invoked from the operating system as needed; this severely constrains the amount of information that can be passed between these processes. Additionally, the necessity of programming the 68000 processor in assembly language severely limits the complexity of programs that can run on it.

2.1.2. Example 2: CMU NavLab

Carnegie Mellon University developed the NavLab explicitly to experiment with outdoor navigation. The NavLab consists of a converted van with several on-board processors. The majority of the computing is performed by Sun-3's connected in an Ethernet network with a WARP parallel computer for high-speed parallel processing[40].

![Block Diagram of CMU NavLab](image)

This system represents a very advanced mobile robot architecture. It has several advantages over the Hermies system previously mentioned. It has a higher level operating system (Unix) which provides advanced debuggers, high level languages and
development systems. Furthermore, the network system provides a higher data bandwidth between processors (up to 10Mbps). Also, the WARP computer provides high-speed (over 10MFlops) parallel processing. Because of the higher overall software and system performance, CMU is able to experiment with creating more advanced distributed control systems; the most important aspect of their work is the hierarchical path planning architecture they implemented, which we will detail later. One should note that this system was pieced together from existing computer systems, the same as the Hermies and most other existing autonomous mobile robots.

Although this architecture is highly advanced and is one of the most complicated mobile robots ever built, the NavLab's computer system has several shortcomings. One of its most significant shortcomings is that it lacks a high-speed bus. Although the bandwidth of the Ethernet is large enough for transmitting commands and general data, it is not capable of handling the data rates required for real-time vision. Furthermore, the I/O bandwidth of the WARP computer makes it difficult to transfer information quickly enough to achieve real-time speeds. Finally, neither the Sun nor generic Unix is particularly suited for the data and I/O intense operations required in a multi-sensor mobile robot.

2.1.3. Example 3: Rice-obot I

The Rice-obot I is an autonomous mobile robot under development at Rice University. The Rice-obot was designed to be a testbed for various aspects of mobile robotics, robotic manipulation and high-level control. One of the most important features of the Rice-obot is its hardware and software modularity; to support this capability, it uses high-level operating systems and programming languages.
There are many on-board processors; the two main processors are a 68030-based Unix processor and a custom LISP machine. These are connected using an Ethernet network.
The Rice-robot uses ultrasonic sensors for coarse obstacle information. In addition, it has a real-time vision system consisting of several vision processing cards connected by a high-speed bus. This bus is completely separate from the VME bus, and thus it does not use up the data bandwidth of the VME or the Ethernet. All of the motion is controlled by separate low-level controllers; this allows the Unix machine to concentrate solely on high-level control and communications. The Unix processor uses a special implementation of Unix designed for real-time control. It has additional capabilities not possessed by normal Unix such as built-in semaphores, shared memory, a very fast interrupt response time, and an interruptible kernel to make it especially powerful for real-time applications[70]. The LISP machine is responsible for the majority of the high-level processing. Both processors include several high-level debugging tools, as well as advanced communications software. Additionally, the on-board network can be easily connected to the campus network to make use of various parallel processors available at Rice. Thus the robot provides an ideal environment for experimenting with complicated control architectures and distributed processing. As with the other mobile robots introduced, the computer architecture uses existing hardware and software with custom modifications for an IAMR environment.

2.1.4. Characteristics of these architectures

There are several general comments that can be made about IAMRs from analyzing the previous robot architectures. Most of these features directly determine how a hierarchical path planning system could be implemented on these robots, and how effective it would be once installed. The major points we wish to emphasize are the following:
1. IAMRs usually include several on-board processors. These processors may be very different from each other, and they may run different operating systems or even no operating system at all. IAMRs may have an on-board parallel processor, but usually it is constrained by slow I/O connections.

2. Most mobile robots do not have custom operating systems. Usually, they use common systems with extensions for whatever communications and control is necessary. Thus they are not usually optimized for mobile robotic or real-time applications.

3. There is often more than one kind of bus or network connecting the processors. These usually vary from being very slow (i.e. serial lines) to relatively fast (i.e. Ethernet). There may also be an ultra-high speed bus for specialized tasks such as vision. The variance in connections also implies a large variance in communications software and protocols.

4. The most common sensors are ultrasonics, but they provide only very rough information. LIDAR sensors are much more useful, but they are also very expensive. Vision systems are not well developed on mobile robots.

We can see from these characteristics that the hierarchy we create will need to be easily ported to many different kinds of computer operating systems and hardware. Furthermore, it will have to handle many different kinds of inter-processor connections, and it must be aware of the inequalities between the data rates of the connections. The data bandwidth of the networks and busses are a limiting factor in data transfer, thus it should not add a large amount of overhead to the communications. Finally, the hierarchy
should not be designed around a particular type of sensor. Instead, it should be relatively easy to replace one sensor with another, and it should use multiple sensors together when possible.

2.2. Basic Path Planning Strategies

Path planners typically use a single representation for the whole working environment. We will refer to these techniques later as all-in-one planners; this reflects the fact that they use only a single all-encompassing algorithm. They can be either classified as sensor-based, in which they use sensors to provide a picture of the obstacles around them, or map-based. The oldest techniques are map-based, and they use sensors only in a limited fashion. All of these planners create some model of the free space, which is just the area between the obstacles, and then generate a path from the start to the goal positions within this space. The map-based planners usually use a graph searching technique or a potential field technique, while the sensor-based planners use a quadtree for representing the workspace.

One concept that will appear frequently in these path planners is the idea of configurations space (C-space). Each point in configuration space represents a particular position and orientation of the robot. Thus a point in C-space uniquely determines the state of the robot. Assuming that we have a robot with some real size navigating in an environment filled with obstacles, if we transform the robot and the environment into C-space, the problem now becomes a point-sized robot navigating through a new environment of obstacles. The new boundaries of the obstacles are calculated by determining which positions and orientations of the robot in C-space would touch the edge of the obstacle in regular space. Let's assume that we have a test robot that can move in the X- and Y- directions independently but cannot rotate. In this case we have a
two degree of freedom robot navigating in a two dimensional map environment (lets assume a two dimensional map for simplicity); the C-space representation might look like the following:

![Diagram](image)

**Figure 5.** Robot in Normal space and Configuration space

In this example, the C-space has two dimensions just as in the real environment. If we allow the robot the additional ability to rotate, then the C-space becomes three dimensional. In general, C-space is used to reduce the robot to a point and then solve the planning for this point. It also has the advantage that rotational motions of the robot can be analyzed along with translational. Usually researchers use C-space assuming an omni-directional robot able to navigate in X, Y, and orientation independently, thus three degrees of freedom. However C-space can also be formulated to restrict the robot in its range of motions; real robotic bases are usually restricted to two degrees of freedom. One major drawback to C-space is the computational time required to transform the environment and the robot into this representation. In general, a path planner which uses
C-space techniques will take an order of magnitude longer than one that does not. Furthermore, C-space is not amenable to sensor-based models.

2.2.1. Example 1: Graph Searching

The graph searching path planners create a graph of the free space and find a valid path by searching this graph. Voronoi graphs constitute the most common scheme for representing the free space. A Voronoi diagram is generated by first calculating the set of points that satisfy the following criteria. We define the distance from a point \( p \) to an obstacle \( O \) as the Euclidian distance from \( p \) to the closest point on the boundary of \( O \); we’ll call this point \( q \).

\[
\text{distance}(p, O) = \min\{ d(p, q) \mid q \text{ element of boundary}(O) \} \quad (1)
\]

where \( d(p, q) \) is the Euclidian distance from \( p \) to \( q \) [29].

Using this, we can create the Voronoi points by generating the set of all points between adjacent objects that satisfy the following property:

\[
\{ p \mid \text{distance}(p, O_1) = \text{distance}(p, O_2) \} \quad (2)
\]

Thus the Voronoi points are simply the points that are equidistant from adjacent obstacles. A picture of these points might look like the following
The Voronoi nodes are defined as the point at which three or more lines of the Voronoi diagram meet (see the figure). The Voronoi edges are the sets of points between nodes; if the obstacles are modeled as polygons, the Voronoi edges will be line segments. However, with more general obstacles, the Voronoi edges will be curves. Usually, the edges are reduced to a series of line segments to simplify calculations. The combination of the nodes and the edges are stored as the Voronoi diagram. These line segments become the line segments for the robot to traverse; and the problem of finding a path is reduced to searching these line segments for a continuous path from the start point to the end point. Usually either the starting point or the ending point or both do not lie directly on the Voronoi diagram; in this case, an additional line segment is added to connect the desired point to the nearest point on the diagram[29].

The obvious advantage to using Voronoi diagrams over other techniques is that once the Voronoi line segments are generated for a given environment, the number
of paths to search is very small. Thus this algorithm is very fast for computing new paths through a known environment. One disadvantage, however, is that these diagrams do not handle dynamic obstacles easily. The computational time for creating the Voronoi diagram itself is very long (usually over 10 minutes) and thus it cannot calculate new paths around moving obstacles. Furthermore, it gives only one valid path between any two obstacles. Thus even if there is a very large amount of space between the obstacles, a small object directly in the path of the robot will completely block the robots' progress. Usually researchers implement a set of support systems that handle the dynamic obstacles and local obstructions. In this way, the Voronoi diagram handles obstacles which never move (such as walls, desks, etc) very quickly, while the support routines handle everything else (such as chairs, dogs, balls, etc.).

2.2.2. Example 2: Artificial Potential Fields

In the artificial potential field techniques, the planner establishes an imaginary gravitational field around each obstacle. These fields deflect possible paths away from the obstacle and into free space. Optimizing the path through these fields places it in a minimum potential energy; this implies (most of the time) that the path avoids the obstacles. C.W. Warren, for example, presents an algorithm using this technique that first creates an arbitrary path through C-space and then optimizes this with respect to the potential fields[27]. In his technique, each obstacle has two components to its potential field; the potential field felt by a point within the obstacle and the potential field of points nearby the obstacle. If a potential path passes through the inside of an obstacle, the potential field it feels should push it strongly to the outside of the obstacle. Since the path could intersect the obstacle from every direction, we want to create a potential that pushes it outward no matter which direction the path intersects the obstacle. The following equation achieves this goal.
\[ U_{in} = U_{max} (1 - \frac{R_{in}}{R_{max}}) + U_{offset} \]  \hspace{1cm} (3)

Where:
- \( U_{in} \) is the potential field inside the obstacle
- \( U_{max} \) is the maximum potential field
- \( R_{in} \) is the distance from the given point to the centroid
- \( R_{max} \) is the maximum radius from the centroid
- \( U_{offset} \) is a base displacement for the inside potential

This forms a cone placed at the centroid of the obstacle with a vertical displacement of \( U_{offset} \). This forces the path uniformly away from the centroid regardless of its orientation. We would also like a potential field outside of the obstacle; this field pushes the path away from the obstacle to avoid collisions due to errors and uncertainties in the robots' position. The potential field outside of the obstacle is given by:

\[ U_{out} = .5 U_{offset} \left( \frac{1}{1 + R_{out}} \right) \]  \hspace{1cm} (4)

Where:
- \( U_{out} \) is the potential field outside of the obstacle
- \( U_{offset} \) is the base displacement for the inside potential
- \( R_{out} \) is the radius from the given point to the centroid.

Equations 3 and 4 are then summed to give the overall potential field. Notice that this external potential field is much lower than the one inside the obstacle; this insures that
moving the path to the outside of the obstacles takes precedence over the distance of the path from nearby obstacles. A picture of this field would look like the following:

![Figure 7. Potential Field of Obstacle](image)

In the technique presented by Warren, the global path is minimized with respect to the potential energy of these fields, and with respect to an additional potential which minimizes the length of each segment[27], producing a minimum length path. Techniques of this kind work very well at producing an optimal path within a minimally cluttered environment. However, when the number of obstacles significantly increase, or the obstacles have very complicated shapes, these algorithms become very slow. Also, potential field techniques have the tendency to fall into local minima instead of global minima. This can cause the robot to get stuck in cul-de-sacs with no way of reaching the goal position.
2.2.3. Example 3: Quadtrees

Quadtrees are a common representation technique used in vision and related fields. They represent a scene by dividing it into four quadrants. A quadrant can be either full, mixed or empty. A quadrant is full if it is completely filled by an object (or a part of an object). A quadrant is mixed if it is partially filled by an object. And it is empty if it doesn't contain any object. A mixed quadrant may be, in turn, subdivided into four subquadrants; notice that it is unnecessary to subdivide a full or empty quadrant because no additional information can be included. These subquadrants may then be similarly subdivided into sub-subquadrants, and so on. This process forms a tree structure in which the root node is the whole scene, the four children are the four quadrants, and the children of these nodes are the sub-quadrants, etc.

![Quadrant Representation](image)

**Figure 8.** Quadtree Representation and Encoding
Notice that a large obstacle usually is subdivided into several different sized filled sub-regions. A valid path through the region would only occupy empty squares. The path planning algorithm searches the quadtree for a set of contiguous empty regions that connect from the start location to the goal. Noborio et al. implement such a path planner using a depth-first search of the quadtree[26]. Their algorithm produces paths like the following.

![Quadtree Generated Path](image)

**Figure 9. Quadtree Generated Path**

Their algorithm also has a second stage in which they use this quadtree representation, in conjunction with a polygonal model of the robot, to determine any rotations that are needed to avoid collisions. In this part of the algorithm, the quadtree is used to quickly determine the closest points between the obstacle and the robot; this information is then used to form a force vector using techniques similar to the potential field techniques presented above. This causes the robot to rotate (if necessary) to avoid the obstacle[26]. Thus we see that the same representation can be effectively used for both determining the
global path and also for making localized corrections for rotation or other constraints on the motion.

2.2.4. Sensor-based navigation

The path planning systems we have heretofore mentioned are largely based upon pre-encoded models of the obstacles and environment. Thus these techniques handle dynamic and unexpected obstacles only in a limited manner. There are another set of techniques that are totally sensor-based. In these techniques, the workspace model is created directly from the sensor readings. Usually, a single snapshot of the environment is taken using video, ultrasonics, or most commonly LIDAR and this is used to generate a model of the obstacles. LIDAR-based systems tend to have the best performance, so we'll look at a system designed by [31] as an example system. In this system, a single scan is taken of the space in front of the robot. A LIDAR scan consists of a discrete array of points (usually 128x128) where each point represents the distance from the scanner to that point in the environment.

![Figure 10. LIDAR Scanner on Robot](image)

This scan is first processed to interpolate between the points, and then converted into a top-down view of the environment. This top-down model is similar to the map-
based models used by the previous planners, except that it includes occluded regions. These regions may or may-not be occupied; either some obstacle in the region is blocking this space from the sensor, or the region is outside the sensor range. The path planner usually handles the occluded regions either by assuming that they are occupied, or by navigating to an intermediate point at which the occluded region can be seen by the sensor. In this particular example, the top-down model is simply an array of pixels, where each pixel represents the height of that point above the ground; the free space is simply the set of points with height zero. Any of the techniques presented earlier can then be used to generate the path.

2.2.5. Capabilities and Limitations of These Techniques

Having examined several different basic path planning techniques, we now wish to draw some conclusions about their capabilities and limitations. These path planners generally lie at one of two extremes: either they make very little use of sensors, or they are totally sensor-based. Thus the map-based systems do not interact with the real environment well, and the sensor-based systems (such as the LIDAR example we presented) cannot determine long-range paths. As a rule, sensor and non-sensor based systems use distinctly different representation techniques. However, because of the scalability of the quadtree representation, it can be used for both kinds of planners fairly efficiently.

Another important factor in differentiating these planners is the computational time required for determining a path. The map-based algorithms that assume a point-sized robot usually take on the order of a minute of computations on a mid-range workstation or 68020-based Unix machine. More advanced techniques which use C-space or a polygonal model of the robot may take up to ten minutes. The LIDAR technique presented takes several minutes to compute the top-down map, and about 30
seconds to compute the path. If the robot is expected to maintain a reasonably high rate of travel, then it must be able to navigate some part of the path at the same time it is computing another part. None of the systems above are designed with this capability.

2.3. Hierarchical Path Planning Techniques

In contrast to the previous path planning techniques, several researchers are now working on handling the overall path planning of an IAMR by creating a hierarchy of planners. Each of the levels of the hierarchy handle navigation through a different part of the workspace; all of the levels cooperate to produce the overall path. Each level may have its own model of the environment or it may share a representation with one or more other levels.

2.3.1. Basic Theory and Structure

Hierarchical path planning techniques that have been developed thus far are based upon an analysis of the techniques humans use to plan their actions. All of these techniques share many common features[48]. A typical hierarchy for robot planning might look something like figure 11. In this system, the user interacts directly with the Mission Planner level of the robot. This level of the hierarchy has the responsibility of establishing goals to achieve the mission objectives. A mission may involve establishing goal points for navigation, grasping instructions for robotic arms, or possibly assembly instructions for manipulators. The mission planner may make use of knowledge bases or other Artificial Intelligence systems to break a high-level task down into low-level goals. For the navigation system, the Mission Planner provides the goal positions and any other constraints (such as time, maximum velocity, etc.) that govern the path planning. In human terms, this level mimics the process humans use to break an abstract task such as
"take out the garbage" down into a series of direct goals (move to point A, pick up item B, etc.)

![Diagram of Classic Planning Hierarchy]

Figure 11. Classic Planning Hierarchy

The Navigator level is responsible for creating a global path between the robot's current position and the goal position given to it by the Mission Planner. A *global path* usually extends well beyond the sensor range of the robot; if the robot is navigating
within a building, the global path might involve moving from a position in one room to a position several rooms away. If the navigating environment is outdoors, a global path may be several miles long. As such, the navigator usually depends on internal maps detailing the environment. A good comparison can be made between this Navigator and the navigator on an airplane. Just as with the mission planner, the Navigator makes only limited use of sensors. However, it receives updates and corrections to its model indirectly from the Pilot level.

The Pilot level of the hierarchy is responsible for navigating through a cluttered environment. Because the Navigator's maps usually do not include all of the obstacles in the environment, and because there may be new or rearranged obstacles, the Pilot level must handle many unexpected circumstances. As such, the Pilot usually makes extensive use of sensors to determine the exact nature of the obstacles around the robot. The environment that the Pilot works within is the local environment, and the Pilot generates a local path. The local path usually extends approximately to the maximum sensor range. If the Pilot cannot navigate the local path, it signals the Navigator that it has failed. Also the Pilot may send update information for the Navigator's maps. In general, many different kinds of communications may occur between the different layers of the hierarchy. The exact nature of these depends on the particular implementation.

The final layer of the hierarchy, the Controller, executes the local path. This layer directly corresponds to classic device controllers. This controller uses the position sensors on the mobile base to measure the difference between the commanded motion and the actual motion; with this information, it uses standard control techniques to minimize the positional error. Since the issues and techniques in this level are very well understood using classical control techniques, we will assume that this level already exists.
2.3.2. Example 1: CMU NavLab

The NavLab uses a planning architecture almost identical to the general model described above. The NavLab’s work environment consists of an outdoor terrain with roads, houses, obstacles, etc. In this hierarchy, the Mission Planner, Navigator, Pilot and Controller levels are referred to as the Captain, Map Navigator, Pilot and Helm respectively. The planning hierarchy looks like the following:

![Diagram]

Figure 12. CMU NavLab System Architecture

In his paper on the NavLab’s navigation system, Goto et al. [40] detail the roles of each of these modules as follows:

**CAPTAIN** executes user mission commands and sends the destination and the constraints of each mission step to the **MAP NAVIGATOR** one step at a time, and gets the result of mission step.

**MAP NAVIGATOR** selects the best route by searching the Map Database, decomposes it into a sequence of route segments, generates a route segment description which includes objects from the Map visible from the route segment, and sends it to the **PILOT**

**PILOT** coordinates the activities of **PERCEPTION** and the **HELM** to perform local navigation continuously within a single route segment
PERCEPTION uses sensors to find objects predicted to lie within the vehicle's field of view. It estimates the vehicle's position if possible.

HELM gets the local path plan generated by the PILOT and drives the vehicle.[40]

Their system varies slightly from the standard paradigm by interweaving some of the control aspects of the Controller into the Helm, Perception and Pilot modules. Each of these modules resides on a separate Sun processor, so all communications between modules occur over the Ethernet. The most interesting element of this architecture is the CODGER system; CODGER stands for COmmunications Database with Geometric Reasoning. The CODGER system is a distributed database facility with additional routines to handle geometric transformations. It acts as the main distribution point for information and commands between the various levels of the hierarchy. Because it is a distributed database, it includes mechanisms for insuring data consistency, for searching for data items, and for coordinating simultaneous requests. The basic data item in CODGER is a token. A token consists of an attribute-value pair. When a module wishes to look up information, it specifies a search expression consisting of a boolean equation using attribute values. All of this parallels the theories developed in distributed databases. The only additional capability CODGER includes over distributed databases is the ability to handle geometric reasoning. The geometric reasoning of CODGER is simply a set of subroutines designed to make transformations between coordinate frames.

2.3.3. Example 2: Brooks' Layering

Although most hierarchical planners have followed a subdivision similar to above, some researchers have taken a totally different approach to the layering. A
notable example of this is the layering proposed by Rodney Brooks[38]. In his approach, each layer adds a more advanced task planning capability to the previous layers. Each layer may directly receive information from the low-level sensors, and it may directly command the actuators. The more advanced layers have the ability to override the commands produced by the lower layers and replace them with their own commands. This system resembles an evolutionary system in which the lower levels of the hierarchy have "primitive" capabilities, and each new layer is more advanced than the previous ones. The hierarchy looks something like this.

![Diagram of Brooks' Layering Technique](image)

**Figure 13. Brooks' Layering Technique**

Brooks' decomposition does not follow the Navigator, Planner, Pilot hierarchy presented earlier. Instead, several layers of Brooks' system work together to achieve the same results as one layer of the classic hierarchy. Because of the kind of subdivision he chose, each layer of Brooks' hierarchy tends to be significantly less complex than similar layers in other systems; Brooks' system compensates for this by having more layers. In some of his mobile robots, Brooks implements these layers using only very simple microprocessors[38]. Brooks' technique has several significant features. Firstly, it
points out that the common perception of how people perform tasks is only one possible subdivision of the problem space. Secondly, it emphasizes the importance of modularity in a hierarchy. And thirdly, it demonstrates that a system which exhibits complicated behavior does not require a complicated structure.

2.3.4. Capabilities & Limitations of Examples

If we compare the performance of the previous non-hierarchical path planning techniques to these hierarchical ones, there are several advantages to the hierarchical techniques. The most important advantage of the hierarchical systems is their ability to handle a much more complicated work environment. Having multiple representations and algorithms tends to strengthen the performance of the overall system by overcoming the shortcomings of a particular representation or algorithm with the strengths of the others. Also, the hierarchical systems make better use of multiple processors and parallel processing to improve performance. By subdividing the problem into levels, we have automatically defined a means of parallelizing and distributing the problem. However, subdividing the problem also adds complexity and overhead to the software. The most significant area of complexity resides in the software for passing information between and among these levels. Information should be easily passed between any levels, communications between multiple processors should be transparent, and the system should encourage a high degree of modularity. The CODGER system is a reasonably good example of a general communications system. Although CODGER at first appears to be simply a distributed database system, it can also be viewed as a simple form of a distributed blackboard. Blackboards are a technique for creating a reasoning environment for multiple processing modules that provides the communications, data sharing and modularity that are critical to hierarchical planning systems. As such, we will make heavy use of the concepts and techniques in that field.
2.4. Blackboard Systems

As we mentioned in the previous section, blackboard theories will be extremely important to our hierarchical path planner to provide a model for communications and process control between the various modules of the hierarchy. We need to begin exploring general blackboard theories to develop an understanding of the concepts and difficulties involved with blackboards. Furthermore, we shall examine three blackboard systems that have been implemented to solve very different problems. Blackboard theories are relatively new, and there is still much disagreement about the terms, concepts and accepted techniques. This variance in techniques and terminology will become more apparent when we introduce the three example systems.

2.4.1. Basic Theory and Structure

The blackboard model was originally developed as a problem solving technique for Artificial Intelligence systems. The blackboard model provides a framework in which knowledge and processes are organized to solve some problem. There are several such frameworks that are used in fields including Artificial Intelligence, intelligent control, and robotics. Some reasoning techniques are fairly simple, such as forward and backward chaining. These techniques describe how to start from an initial set of conditions and reason till a final set of conditions is satisfied. Blackboard theory provides a much more complete reasoning environment than other systems because it allows a dynamic chain of reasoning to be used to solve the problem. Furthermore, it provides a much more detailed description of the interactions between sources of information.

The easiest way to understand the blackboard model is by using the analogy of several people standing around a blackboard. These people are given a problem to
solve; this problem is usually very complicated and involves many sub-problems and a large domain of knowledge to solve it. We will assume that the problem is so difficult that no one person can solve all of it, although there is enough knowledge distributed among the people to produce the answer. We can further assume that each person has a specialty topic in which he is an "expert". And each person will use his expertise to solve some part of the problem. Now what we need to define is a means by which each person communicates with the others. The most obvious solution is for each person to talk directly with the others. This technique does not provide for good modularity; if some people leave or new people arrive, it is hard to readjust the communications structure. Instead, we will assume that all of the people are sitting in a room looking at the blackboard. When the problem is introduced, it is written on the blackboard. Any person (or persons) may then go up to the blackboard and write his theories down about the solution to some part of the problem. After this person has written his theories, other people may use their expertise to either refine or replace some of these theories with their own. Notice that not everyone will understand every new theory on the board; usually a particular person will only understand the theories relating to his expertise. Once they have written their theories, other people come up and write their theories. This continues until either the solution is found, or some limiting constraints have been reached (such as maximum time). We notice that in this system, we can make use of both forward and backward chaining, as well as other techniques, in different phases of the problem solving.

Now we can formalize this example to create the blackboard model. The people in our example represent sources of knowledge and inference. In the model, these are referred to as Knowledge Sources (KS); a knowledge source corresponds to a combination of inference engines and knowledge bases commonly used in other AI techniques, such as expert systems. Thus Knowledge Sources include information as
well as \textit{algorithms}. All of the Knowledge Sources communicate using a common working memory called the \textit{blackboard}. The Knowledge Sources know how to access various pieces of information from the blackboard and write information (i.e. theories) to the blackboard; they do not know what other Knowledge Sources exist or how they use the information on the blackboard. From their point-of-view, only they and the blackboard exist. Whenever a KS writes information to the blackboard, it writes it in a particular format. This format may be known by several Knowledge Sources. There may be multiple pieces of information on the blackboard with a particular format; this corresponds to having multiple theories about some aspect of the overall problem\cite{2}. A block diagram of this would look like the following.

\begin{center}
\begin{tikzpicture}
  \node (blackboard) [draw, rounded corners] {Blackboard};
  \node (source1) [draw, rounded corners, below of=blackboard] {
    \begin{tabular}{ll}
      Inference Engine & Knowledge Base \\
    \end{tabular}
  };
  \node (source2) [draw, rounded corners, below of=source1] {
    \begin{tabular}{ll}
      Inference Engine & Knowledge Base \\
    \end{tabular}
  };
  \node (source3) [draw, rounded corners, below of=source2] {
    \begin{tabular}{ll}
      Inference Engine & Knowledge Base \\
    \end{tabular}
  };
  \draw [->] (blackboard) -- (source1);
  \draw [->] (blackboard) -- (source2);
  \draw [->] (blackboard) -- (source3);
\end{tikzpicture}
\end{center}

\textbf{Figure 14. Blackboard and Knowledge Sources}

This model effectively subdivides the problem space into a hierarchy of modules; the particular organization of the hierarchy depends upon the application, and it can be changed. When multiple theories are on the blackboard at the same time, we need to have some kind of arbitrator to judge the order in which the possibilities should be
pursued; similarly, we might have several Knowledge Sources which are capable of investigating a particular theory and only one should be invoked at a time. If this arbitrator didn't exist, it is possible that the blackboard would spend forever investigating incorrect theories. This arbitrator is generally referred to as the scheduler. The scheduler is responsible for determining which Knowledge Sources should be invoked, and in what order; schedulers vary from being very simple, to enormously complex[2].

In general, this is a very simplistic model; there are usually numerous complications to corrupt this model. For example, in a real processing environment it may only be possible for one Knowledge Source to access the blackboard at once. Also, the Knowledge Sources may be distributed among separate processors. These and other problems tend to make the actual implementations much different from the model. However, the model still provides a straightforward framework for analyzing and comparing various problem solving systems.

2.4.2. Example 1: Hearsay II

One of the first programs to use a blackboard was the Hearsay II system. Hearsay was designed at Carnegie Mellon University as part of a DARPA project on natural language understanding[1]. Hearsay involved several competing and complementary Knowledge Sources. In this implementation of a blackboard, each Knowledge Source specifies the conditions under which it should be activated. Whenever that set of conditions occurs, the KS is activated and formulates a hypothesis using the information on the blackboard; this constitutes an interrupt driven system.
Each of the Knowledge Sources has a particular task. These duties range from segmenting the signal to determining when to stop the calculations[1]. Within this system, the designers opted to use both forward and backward chaining reasoning in different parts of the problem solving process to improve the performance. Although Hearsay did not completely achieve the results originally desired, it represents a significant achievement, and it was the first system to demonstrate the feasibility and advantages associated with using a blackboard architecture in solving problems with a large problem domain.

2.4.3. Example 2: BB1

The BB1 system is a much more recent blackboard system; it is an attempt to produce a general blackboard environment in which many different kinds of intelligent systems can be created. As in the Hearsay system, BB1 is a blackboard system in which KS access pieces of information. In this system, the blackboard contains data items called *objects*; the programmer can specify dependencies between objects as well as other
special attributes (such as procedural vs. static objects)[4]. The Knowledge Sources can then access and modify these objects; modifying a single object may affect other objects if the other objects are dependent on the modified one. The BB1 system has an additional blackboard specifically for scheduling called the control blackboard. The implementer can create so-called control Knowledge Sources that determine criteria for scheduling. When the scheduler needs to schedule Knowledge Sources in the blackboards, it uses the criteria in the control blackboard to determine which process to schedule. In this way, different reasoning techniques (i.e. forward, backward, etc.) can easily be interchanged throughout the various stages of the problem[4].

BB1 provides a rich environment for development and reasoning; however, it is limited by several factors. Its first limitation is a by-product of its strengths; BB1 is a development tool and as such it does not contain any intelligence itself. Although it includes the complicated control blackboard for scheduling and numerous options, it is the implementer's responsibility to effectively make use of it. And the complex user interface and scheduler add a large amount of overhead. This makes the blackboard very slow and very large. We have already indicated that for the task of hierarchical path planning, we need an extremely fast and relatively small system; thus this system would not be appropriate to our needs.

2.4.4. Example 3: BLOBS - An Object-Oriented Blackboard

Another interesting Blackboard Architecture is the BLackboard OBjectS (BLOBS) system developed by Zanconato et al. This architecture has several distinct features from the previous blackboards. The most significant difference between this architecture and the previous ones is its reformulation of the blackboard model to combine the data items of the blackboard and the Knowledge Sources into a single object-oriented object[9]. These objects include most of the normal features of object-
oriented objects, as well as features for interfacing with the blackboard. In this case, interaction with the blackboard consists of accessing information in other objects. Objects can inherit declarations and variables from parent objects. And new instances of objects can be created with initial values for the local variables. This blackboard architecture was designed with special features for control applications. To facilitate this, the system includes a pseudo real-time clock. This clock is used to control the processing time of the behaviors. It also provides timer capabilities for awakening process at preset times and to provide timestamps for objects.

A significant aspect of this system is the technique it uses for accessing data. If we review the classical blackboard model presented above, we see that it places absolutely no restrictions on which Knowledge Sources can access a particular piece of data. Because of this, it is fairly easy to create the situation in which multiple Knowledge Sources read and write to the same data item simultaneously. Thus general blackboard systems must include some built-in mechanisms to insure data consistency and conflict resolution. The BLOBS system avoids this problem completely by removing the possibility of simultaneous writes to the same data item by different processes. Objects can only write data to their own variables; the variables of other objects are strictly read-only. When an object requests the value of a variable of another object, the blackboard simply makes a copy of that value. Thus this system could easily be expanded to multiple processors. The advantages of this access technique will become more apparent when we look at distributed blackboard systems.

2.5. Distributed Blackboards

A significant advance in blackboards has occurred recently with the implementation of blackboards that function on parallel processors and in distributed
processing environments. A loosely coupled multiprocessor environment places constraints upon sharing data, and the two systems in this section reflect distinctly different approaches for handling this. Most importantly, there is a great difference between communications between processes on the same processor and those on different processors. We will define local communications as communications between processes on the same processor, while global communications occur between processes on different processors. General distributed systems deal with the difference between local & global communications in various ways. In the first distributed blackboard system we will look at, global calls are implemented as a special Knowledge Source which forwards information over the network. This system is classified as a non-transparent system because there is a visible difference between local and global calls. In the second system, there is no visible difference between local and global communications; this system is called a transparent system.

2.5.1. Example 1: Distributed Vehicle Monitoring Testbed

The Distributed Vehicle Monitoring Testbed is a system developed by Lesser etal. for researching the various issues of distributed problem-solving networks; the particular task it models is monitoring vehicles travelling through a large area. Spread throughout the area are sensors for determining nearby vehicles. The area is subdivided into many small regions with a single processing node per region. Using these distributed sensors each region develops an estimate of the paths traversed by each vehicle and monitors their motion as they wander through that particular region. Nodes communicate with each other to converge to a global solution from possibly erroneous intermediate hypotheses; Lesser etal. refer to this type of processing as functionally accurate, cooperative (FA/C)[6]. The blackboard architecture used in this system is largely dependent on the Hearsay-II system. Their distributed environment consists of several
nodes each running a complete Hearsay-II system. The Knowledge Sources of the Hearsay system are modified to handle the signal processing necessary for this particular task. Additionally, the Hearsay systems are expanded to include special *Communications Knowledge Sources* which contain the communications and activation primitives needed to transfer information from processor to processor. Blackboards for control, goals and consistency were also added. The whole distributed architecture looks like the following.

![Distributed Vehicle Monitoring Testbed Blackboard](image)

*Figure 16. Distributed Vehicle Monitoring Testbed Blackboard*

Since global communications only occurs through these special Knowledge Sources, there is an obvious distinction between local and global calls. This type of architecture assumes that there is a distinct difference between the kinds of processing done locally and the information passed globally. In this blackboard, each processor has an area of expertise and only results from these calculations are distributed globally. Thus the subdivision of the problem in this system is static across processors. Because of the distinction between local and global communications, the throughput of local communications is largely independent from the global communications' overhead. This is significant because most of the transactions to the blackboard will be local, and thus the throughput of the local calls is the greatest factor in the overall speed of the system. On the other hand, by selecting the Hearsay-II system for each node, they have included its large overhead.
2.5.2. Example 2: Database Transaction-based Blackboard

As in the previous example, a distributed database system also assumes that the blackboard spans several loosely-connected processors, and some number of Knowledge Sources reside on each processor. However in this system, all accesses to the blackboard appear uniform over all of the processors. In other words, it is impossible to tell whether the data being accessed is on the same processor or a different processor. The blackboard is accessed using *transactions*; these transactions are exactly like transactions in databases[7]. A transaction is simply a series of reads and writes to a data item; a transaction begins with a command to initiate the transaction, and ends with a commit command. The reason transactions are very important in database theories is that they provide a certain level of guarantees to the process that is accessing the data. The first guarantee they provide is data consistency. Usually while one process is writing and reading data to some location, another process should not be allowed to write to the same location. If multiple processes read and write to the same data item simultaneously, inconsistent values may be stored in that data item. We can clarify this problem by presenting an example. In this example, we assume two processes X and Y each perform a series of reads and writes to the variable A. Suppose process X wrote a 1 to variable A, read the variable, and then wrote a 3 to A. And suppose that process Y wishes to read the value of A, and then write 5 to A. Depending on the exact interweaving of these reads and writes, different results will result. Suppose A starts out with the value 0.

<table>
<thead>
<tr>
<th>Sequence 1</th>
<th>X Write 1 to A, X Read A (results is 1), X Write 3 to A, Y Read A (results in 3), Y Writes 5 to A (A ends up with 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence 2</td>
<td>X Writes 1 to A, Y Reads A (results in 1), X Reads A (results in 1) X Writes 3 to A, Y Writes 5 to A (A ends up with 5)</td>
</tr>
</tbody>
</table>
Sequence 3  Y Reads A (results in 0), X Writes 1 to A, Y Writes 5 to A, X Reads A (results in 5), X Writes 3 to A  (A ends up with 3)

The most common way a database insures consistency in reads and writes is to lock some of the data during a transaction. If a particular data item is being written to by a process during a transaction, that data is locked from either being read or written by other processes until that transaction ends. If a data item is being read during a transaction, then it is locked from writes by other processes during that transaction. These software locks can become extremely complicated; more advanced locks usually allow greater concurrency in accesses while still insuring data consistency.

The other main assurance that a transaction can provide is atomicity. Atomicity provides that if a process (or processes) crashes during a transaction, either all of the data in the transaction was written to a permanent location, or none of it was. If this condition did not occur, the permanent storage facility (i.e. the harddisk) could get corrupted by having some new data and some old. In the transactional model, when the transaction is completed, the transaction is either committed or aborted. If the transaction is committed, then it is absolutely certain that all of the data was written to permanent storage. If the transactions was, for any reason, aborted, or if the system crashes before committing, then NONE of the data was written. In this way, the system can know exactly what data is new and what data is old after a crash.

The blackboard system of Ensor implements a fairly complicated transaction model to achieve these assurances[7]. The transaction model details the series of steps in a transaction from the start to the end. The state diagram for their model looks like the following:
The Straddle and Precommit states are needed because of the locking mechanism chosen to insure consistency across the multiple processors. Insuring data consistency of a distributed database can become very complicated. And this complication adds overhead to all accesses to the database, not just global accesses. However, as Ensor points out, the greater homogeneity of their system and its ability to handle numerous processors make it much easier to experiment with distributed knowledge systems. Furthermore, this system enable the developer to rearrange the tasks associated with each processor much more dynamically than the non-transparent architecture previously presented.

2.5.3. Summary

In the first blackboard system, the global communications were handled in a very simple manner. This system did not need any data consistency mechanisms because very little data was actually shared, and it did not have concurrent writes between separate processors. Thus the global communications has relatively little overhead. On the other hand, the code for these Knowledge Sources have to differentiate between global and local calls. Thus the Knowledge Sources have to maintain an understanding of which processors are handling what kind of information. This contradicts the whole concept of modularity that make blackboards a powerful problem solving technique. It is very difficult to move the Knowledge Sources from one processor to another in this system;
thus this blackboard is best suited for control applications in which the tasks associated
with each processor are strictly defined.

The transactional blackboard system provides a much greater flexibility. It
allows the implementer to easily experiment with different task decompositions without
requiring any code rewrites; this system is highly modular and uniform. But there is a
definite cost to this uniformity. The greatest cost is in the overhead incurred in every
transaction to the blackboard. It would be much better if the local and global accesses
did not require the same overhead. Also, significant problems can arise in a real-time
system due to the locks on the transactions. If one KS has the lock on some data and
does not end the transaction quickly (and thus holds on to the lock), he may cause
problems with other processes which need quick access to those variables. The best
implementation would allow multiple operations within a transaction. And it is
important that the local accesses do not incur additional overhead due to locking
mechanisms needed for the global accesses.
Chapter 3

Hierarchical Path Planning Architecture

3.1. Subdivision of the Problem space

We discussed in the background section the hierarchical techniques used by other researchers in subdividing the path planning problem. We now wish to develop our own concepts of this hierarchy. We will first introduce a model environment as an example of a problem space in which the robot has to navigate. Then we can compare the efficiency and effectiveness of different subdivision techniques using this example environment to generate some idea of the complexity involved in each technique. Our goal is to subdivide the problem space in such a way as to be able to make use of existing path planners, while greatly expanding their usefulness and flexibility. Thus we want to analyze this model system, characterize good subdivisions of the problem space, and then determine what kinds of existing planners work well in these subdivisions.

3.1.1. Model Environment

We wish to introduce an example environment that will enable us to calculate the computational complexity of varying hierarchies, algorithms and representations. We could simply define the environment as a field containing random objects; but this does not provide us with any realistic idea of the difficulties involved with navigating real
environments. In general, there are two model environments which are good for this task. An outdoor terrain provides a very complicated and dynamic environment for navigation; usually this environment covers several square miles. This environment requires very local planning to get around boulders, trees, and potholes as well as long distance (i.e. "map guided") planning. However, outdoor navigation is very much a three dimensional planning problem due to hills, valleys, embankments, etc. that further constrain the possible paths. It would be very difficult for us to accurately estimate the complexities associated with the representations and algorithms for navigating in this environment because very little is known on how to efficiently model these three dimensional features.

The other environment we can use as a model is indoor navigation through the hallways, rooms, and offices of a building. There are several advantages to using this as our environment. Indoor navigation still includes the necessity of local navigation, to avoid obstacles such as tables, chairs, walls, etc., as well as global navigation from room to room. In general, any path planning which requires the robot to navigate to a position well beyond its sensor range will require both local and global navigation. In addition, the path planning can be modeled as mostly two dimensional. This simplifies our estimates for complexity of algorithms. Additionally it is easy to estimate the complexity of the representations at any particular level of detail.

The model environment we chose is a large building with hallways, rooms, offices, etc. A general task for the robot might be to navigate from office to office depositing mail in the mailbox of each person. A map-based representation for the whole building might look like the following:
In this representation, a certain level of detail has not been included, namely the smaller objects within each room. If we "zoom" in to a particular room of this map, we see a much greater level of detail. By zooming in to produce another map, we have effectively subdivided the problem space into subspaces consisting of single rooms. This is just one possible subdivision of the environment, but it is the most common one for humans.
If we analyze why this subdivision is used, we see that a room roughly corresponds to the sensory limit of a human. Also, the entrances to rooms subdivide the search space of the path planner into distinct subpieces. Thus it forms a natural division. However, there are disadvantages to this subdivision; for example, the size and shape of the room maps vary depending on the size of the room. We could also subdivide the building by "layers"; then we could have planners that knew how to traverse each layer. For example, we could separate out all of the hallways and have a planner that knew how to traverse only hallways. Or we could have one path planner that avoids large stationary obstacles, while another one avoids moving obstacles; in this case the domain of each planner completely overlaps the domain of the other. In general, the domain of each of the planners can be overlapping or non-overlapping. Each subspace imposes constraints
on any other subspaces that adjoin or overlap it. These constraints must propagate through all of the path planners to generate a valid global path.

To get some idea of the number of obstacles the robot is likely to encounter, we can estimate that in a "normal" office (such as the one pictured above) the robot is likely to encounter on the order of 10 large obstacles and 10 small ones. From the example building above, we could estimate the total number of offices at around 50. This means that our example workspace includes approximately 1000 obstacles. Most path planning systems, which usually take on the order of 1 to 10 minutes of processing time, use an example workspace containing a maximum of about 10 obstacles. Thus it is critical that we reduce the number of obstacles that the planner has to keep track of; using a hierarchy will enable us to make significant reductions in the analysis.

3.1.2. Subdivision Techniques

We will analyze three different approaches to subdividing the domain to compare their usefulness.

1. The all-in-one approach

The first subdivision we will examine is a simple single-leveled hierarchy. In this representation, all of the planning is done by a single path planner. Such a system would use a representation which included the whole work environment, including all obstacles, in a single map. Most existing path planners are constructed in this way. By treating the path planning in an all-in-one manner, the planner becomes a straightforward, standalone program. Obviously there are no conflicts between planners (since there is only one), although there still may be competing hypothesis that the planner is considering at any given time. Furthermore by having only one representation
of the environment, the planner itself doesn't have to worry about data fusion and data consistency between representations.

Unfortunately, there are several problems with handling the planning in this way. The first thing we will look at is the computational complexity of this subdivision. Given some workspace which includes $M$ square units of space, we want to analyze how many obstacles the planner has to keep track of. Suppose that there are $k$ obstacles per square unit. Thus we see that the total number of obstacles is given by:

$$\text{Total Obstacles} = T_0 = k \times M$$

(5)

In general, the length of the path between a start and goal point in the workspace will vary in length and complexity depending on the placement of the obstacles. Thus the planner will sometimes produce a valid path in a relatively short time, while other times it may take longer. Furthermore, the actual time to calculate the path will vary depending on the planning algorithm and the way the obstacles and robot are represented. Thus it appears difficult to characterize the general computations required. But regardless of the algorithm or representation used, we can estimate an upper bound on the complexity by realizing that the number of obstacles in the path directly relates to the computational complexity required to plan a path between them. In a particular representation, the complexity of one obstacle will be approximately equal (within an order of magnitude) to that of any other obstacle even though the size and shape of the obstacles will vary. Using this information, we can estimate the total number of edges in the entire environment. Jacobs et al. present a technique for generating a smooth path for a mobile robot[34]; they estimate the computational complexity of the core of their algorithm as $O(n^2)$, where $n$ is the total number of edges of all of the obstacles. Using our assumption above that all obstacles have approximately the same complexity, we can estimate that the total number of edges is given by the following equation:
\[ n = f \times T_o \]  

(6)

where \( f \) is the number of faces per obstacle. If the algorithm takes \( O(n^2) \), then by substitution we show that we can also derive the complexity in terms of the number of obstacles

\[ C_C = O(n^2) = O((f \times T_o)^2) \]  

(7)

\[ C_C = O(f^2 \times T_o^2) = O(T_o^2) \]  

(8)

No matter what representation or algorithm we use, the best we can hope to achieve is linear complexity \( (O(T_o)) \). If we substitute into this our estimate for \( T_o \), we see that the computational complexity is, at best, given by

\[ C_C = O(k \times M) \]  

(9)

In this situation, if we halve the number of obstacles the planner needs to be aware of, we correspondingly halve the computational complexity. In the more realistic case of \( O(T_o^2) \) complexity, by halving the number of obstacles we reduce the complexity by a factor of four. Thus there is a direct tradeoff between the detail of the representation and the speed of the planner. It is obvious that if we use only a single path planner for the whole workspace, this path planner must be aware of all of the obstacles in the environment; otherwise it cannot plan a reasonable path. In the other hierarchies we will examine, we can significantly reduce the number of obstacles each path planner needs to examine, and thus we reduce the computational complexity.

Another problem with this approach is the difficulty in formulating a single representation and algorithm that handles both the sensor information about the dynamic obstacles in the robots' path as well as global navigation from room to room. When we
introduced several different common path planning algorithms in the background section, we noticed that each planner tended to lie at either one of these extremes. Thus it is not clear from these examples what single representation and algorithm could be used to handle the diversity of path planning required.

Another problem with using a single planner is that its response time is usually much more than a minute. If we analyze the response time required for a mobile robot to navigate at a reasonable rate, we see that it needs both a fast response mechanism and a slow one. A single pause before traversing a long path is acceptable as it does not last more than a couple of minutes. But while the robot is traversing the obstacles, it needs to be able to respond to unexpected obstacles very quickly (i.e. within a second). The all-in-one planners presented before have reasonable calculation times for the single pause, but they cannot handle the fast responses required for local navigation. In most cases, when researchers have implemented path planners using those techniques, they have also implemented a mini local planner to handle the fast responses.

2. Classic Hierarchy

Next we will look at the hierarchy used in the CMU NavLab and various other IAMR robots. As we mentioned in the background, this subdivision separates the planning into two levels: one level deals with "map following" and is usually referred to as the Navigator. The other level deals with a "windshield" view of the world and is usually called the Pilot. We will ignore the other levels of the hierarchy they use, since they are not directly responsible for path planning. For consistency, we will refer to the Navigator as the global planner and the Pilot as the local planner. Each level of the planning includes a single planner especially designed for that level of abstraction, and each level has its own model of the environment. Researchers have gravitated to this subdivision because it follows the "logical" subdivision in planning that humans seem to
use, and as we have already shown it has various advantages over the all-in-one techniques. The most significant of these is the separate representations for global and local environments. This generally solves the problems in the previous hierarchy relating having only one representation. Additionally, by separating the local from the global planning, we can greatly improve the response time of the system. While the global planner is calculating the next long-range path, the local planner can be traversing the previous. Thus even though the global planner takes several minutes to calculate a new path, the robot can still maintain a relatively high rate of motion. This also means that unexpected difficulties can be quickly resolved by the local planner usually without involving the global planner.

The other main issue we wish to analyze for this hierarchy is its computational complexity. Suppose that the local planner works within a space that is $1/j$ of the total workspace. The average number of obstacles in this subspace is given by:

$$T_{ol} = \frac{k \times M}{j}$$

(10)

Since the local planner can make corrections for obstacles ignored by the global planner, the global planner no longer needs to keep track of all of the obstacles in the workspace. We can assume that it does need to be aware of some percentage of the number of obstacles, which we will call $P_c$. Thus the global planner needs to handle the following number of obstacles:

$$T_{og} = k \times M \times P_c$$

(11)

If we make the assumption that the local frames overlap each other very little, then we can estimate the total number of obstacles that the local and global planners have to compute
\[ T_0 = \left( \frac{k \times M}{j} \right) \times N + (k \times M)xP_c \]  

(12)

where \( N \) is the number of local frames that are analyzed while traversing a global path. If we look at our model environment, we can generate some approximations for \( P_c, j, \) and \( N \). It is likely that the global planner will need to retain information on some of the big objects in each room, but it won't need to know about the little ones. We might estimate \( P_c \) to be about 25%. Furthermore, an average path might traverse about half of the rooms in the building, giving a value for \( N \) of 25. And since the local planner's range would be approximately equal to the size of an office, we would probably assign the local planner to planning within one room; this means that \( j \) would be 50. If we plug these numbers into the previous equation, we see that the total number of obstacles that the hierarchy needs to analyze is now 750. This gives us a 25% improvement versus the previous all-in-one planner. Assuming both the local and global planners have linear speedup, this gives us a corresponding 25% increase in the overall computational complexity.

The difficulty in this system stems from the overly simplistic model of the environment and robot that the global planner uses. One should first realize that ideally the path passed to the local planner should be navigable as long as no unexpected obstacles are within the workspace. In other words, the global path planner should have an accurate enough representation of the environment and the robot such that the majority of paths it produces are navigable under ideal conditions. If the majority of paths are not navigable, then the robot will most likely never find a valid path. At first, it would seem that any path that the global planner develops should be usually passable. However, many global planners have such a simplistic model of the environment that they cannot take into account many "realities" that constrain a robot's motion. A common example of this is the assumption made by some global planners that the robot
has no real physical size. Obviously most real robots are actually quite large, and many could never fit through a normal doorway. Thus if such a global planner was used to plan a path between two rooms with a single narrow doorway, it would generate a path that always passed through the door. Therefore the global path would be always impossible to traverse no matter what alterations to the path the local planner made. In this particular case, the robot would simply fail to move to the goal point. But in other cases, the arrangement of obstacles might be much more complex, and the robot might wander around the room for a long time before finally giving up. Searching for a valid global path in this way is not optimal for two reasons: the hierarchical system is not well suited for performing the majority of planning in this way, and it causes a large amount of wasted motion.

The simplest solution to this problem is to add the necessary detail to the global planner to ensure a great enough "reality" such that the majority of the paths it generates are navigable. There are two problems with doing this. The first problem is similar to the all-in-one system presented earlier; namely, the computational complexity involved with adding such detail would be so great that it would take an excessively long time for the planner to generate even a simple path. In addition, the representation that such a planner would need to use would be very complicated, even if the majority of the time it navigated through a simple environment. Thus in order to handle a situation that might occur only under limited circumstances, the overhead of all the planning would greatly increase. The other problem with this system is its inflexibility. If the robot were introduced into a new environment with a different kind of constraints (for example, overhanging pipes instead of street curbs), it would be difficult to modify the global planner to take these into account without making significant changes to the planner and representation. The solution to the problem that we adopted takes a different approach. It adds another level to the path planning hierarchy, between the local and global...
planners; this level contains domain-specific planning information that enhances and verifies the global path before it gets passed to the local planner.

3. Our hierarchy

As in the previous hierarchical systems, our hierarchy includes a global planner as well as a local planner. In addition, we include an intermediate level that consists of a number of constraint analyzers. These analyzers are designed to analyze the global path for some particular constraint; this constraint might consist of unusual protrusions that don't appear in the global map, tight corners that require the robot to rotate to fit, or other realities such as a more realistic model of the shape of the robot. If they encounter a situation in which the global path is not feasible, they can either make slight modifications to it, or request a whole new global path. If the global path conforms to all of the relevant constraints, it is passed on to the local planner.

![Three-level Hierarchy](image)

Figure 20. Three-level Hierarchy
This hierarchy has many positive features. The hierarchy ensures that the majority of the paths generated within a particular domain and passed to the local planner will be feasible. There will still be instances when the local path is totally un navigable, such as when a large unexpected obstacle or an unusual one is encountered, but these generally infrequent situations can be easily corrected at the constraint or global levels. Situations such as paths through doorways which are too narrow for the robot will never propagate to the local level. Additionally, the system can be easily customized for different environments and different kinds or mobile robots. For example, if the robot is navigating in a plant, a constraint analyzer could be added for avoiding overhead pipes. Another very useful analyzer could handle maneuvering in tight spaces. Or even better, a constraint analyzer could be written to determine what kinds of constraint analyzers are needed in a particular section of the workspace. Also, we could write a rotational constraint analyzer that handles maneuvering for a non-omnidirectional robot, and another one for omnidirectional ones. In general, adding the constraint analysis level enables the overall hierarchy to handle a greater diversity of real, obstacle-filled environments without adding enormous complexity to the representations.

Assuming that we have a hierarchy of this type, we would like to determine some kind of estimate of its computational complexity in comparison to the other techniques presented. Achieving an estimate is significantly more difficult than in the previous examples because the complexity of the analyzers varies considerably depending on what constraint they are analyzing. Furthermore, much of the improvement in computational complexity occurs indirectly due to the simplification possible in the global model. For example, suppose our workspace included overhead pipes, and we wished to expand the global planner to handle these. The robot might sometimes carry supplies over its head which make it too tall to fit under these pipes, while other times it could squeeze through.
Since most of the global planners use a two dimensional model of objects, we would need to expand this to add an understanding of elevation above the floor, which would entail a significant increase in the computational complexity associated with planning around each object. Looking at our model environment, we could estimate that this environment might contain 10 rooms with low doorways or ceilings. If we handle these obstacles separately, this added computation would be equivalent to the computation time of 10 additional obstacles, which is only a small fraction of the total number of obstacles. However, if we modify the representation of all the obstacles, then we would most likely double the complexity of planning around each obstacle, and thus at least double the overall computations.

3.2. Details of the Global Planner

Having developed a general subdivision for our hierarchy, we can now detail the role of each level. We will begin by describing the role of the global planner and we will develop some criteria for deciding what kinds of path planners work well in this category. Once we have this criteria, we can use it to compare the usefulness of some of the most common planning techniques for this role. Finally, we will detail an example planner that we used to physically experiment with this level of the hierarchy.

3.2.1. General description

The global planner in our hierarchy has the role of determining long-range paths well beyond the sensor limits. In our model environment, this would encompass generating paths from room to room and a general knowledge of the large obstructions in each room. The representations for this level only need to include the larger objects, such as desks, walls, etc.. The paths generated can be optimized, but at this level of
abstraction it is more important to maintain a safe distance from obstacles than to follow the shortest route. And the response time for this system can be very long, up to several minutes. Another criterion for the global representation is that we should be able to easily extract subspaces from the model to distribute to lower-level planners. In general, the global planner should not worry too much about having a highly detailed model of the environment.

3.2.2. Types of Representations and Algorithms

Many of the classical path planning algorithms are useful as global path planners. Any of the Voronoi-related techniques will work. These tend to use a polygonal representation for obstacles. For example, if we look at the obstacle representation used by Meng [29], we see that their representation uses two dimensional polygons to represent obstacles. Depending on the number of faces in the polygon, the resolution of these techniques may be as much as a few inches, although usually they are not this precise. This algorithm generates a path in much less than a minute, and so is fast enough to provide a reasonable response time. Furthermore, it is easy to calculate which obstacles appear within an arbitrary subregion; this information can be used to generate local maps. Meng’s planner makes the common assumption that the robot has zero size. Another effective representation for this category is quadtrees. The system by Noborio we introduced earlier [26] is significantly more complicated than Meng’s Voronoi-based planner insofar as the robot is represented as a polygon, and the planner can handle rotations of the robot for more complicated maneuvering. It takes this system on the order of five to ten minutes to determine a global path; however, this time is greatly dependent on the number of obstacles in the environment. If we use this system in our hierarchy, we could improve the speed by decreasing the number of obstacles in the global map. And since the representation recursively subdivides the workspace in the
process of planning, it can provide a sub-map for the constraint analyzers without additional computations. There is a tradeoff between using this algorithm and the more simplistic Voronoi one. If the workspace is relatively uncluttered, we can assume that most of the paths of the robot will not require rotations, and we can also assume that most of the paths generated assuming a point robot can, with little or no modification, be traversed. In this circumstance, it is better to have the simpler, faster Voronoi system as the global planner, and handle rotations at the constraint analysis level as needed. If, on the other hand, the workspace is much more heavily cluttered, we can assume that most paths will require complicated rotations. In this case, by handling at least the coarse rotations at the global level, we minimize the number of unnavigable paths that the constraint analyzers have to reject. In this case, there is a net gain in performance even though the global planner is slower. However, by installing the rotation at the global level, we have lost the flexibility to handle many different kinds of mobile robots. In our model environment, we can assume that most of the global paths will be passable, and little rotation will be required. In this case, it is preferable to use the simple point-model of the robot and handle rotations as special situations. Most of the techniques that have been classically used as all-in-one planners can work reasonable well as a global planner.

3.2.3. Example Global Planner

We used a Voronoi-based path planner created by Alex Meng from Texas Instruments to experiment with the global planner in this hierarchy. Meng's program was designed to navigate in a factory environment with large spaces and fixed, large obstacles[30]. The Voronoi technique works well in this kind of environment. Given a particular workspace, Meng's algorithm first generates the Voronoi paths. When the operator inputs a goal location, the system calculates a path from the current robot
position to that goal using a retraction technique. To find the path, Meng's algorithm first uses retraction to generate the initial segment to connect the beginning and endpoints to the Voronoi graph.

![Retraction of point S](image)

Figure 21. Retraction Planning

The simplest way of finding the retraction segment (i.e. s to q) is by finding the closest point p between the obstacle A and the point s, and then finding the intersection of the halflne segment ps to the Voronoi graph (i.e. q). Having generated these segments, Meng's algorithm now searches the Voronoi diagram for a path from the start to the goal. The technique he uses for searching is the Dijkstra's shortest path search. The algorithm Meng uses is described algorithmically as follows.

Input A set lambda of obstacles O defined by their boundaries
A Voronoi graph G = (V,E) defined by lambda (V= vertices, E=edges)
The start location s and goal location g
Output A collision-free path from s to g
If \( s \) and \( g \) are in \( V \)
THEN Use Dijkstra's shortest path search on \( G \) from \( s \) to \( g \) 
and return the path

IF \( s \) or \( g \) is on an edge of \( E \)
THEN Adjoin \( s \) or \( g \) as a new node of \( G \)
  Split the edge which \( s \) or \( g \) is on into two subedges
  Use Dijkstra's shortest path search on \( G \) from \( s \) to \( g \)

LET \( O_1 \) be the closest obstacle to \( s \) in lambda
  \( A_1 = \{ p \mid p \) is a nearest point to \( s \) on Boundary(\( O_1 \))\}\)
Choose one point \( p \) from \( A_1 \)
\( s \)-bar is the intersection point between the half line
  from \( p \) to \( s \) and an edge in \( E \)

LET \( O_2 \) be the closest obstacle to \( g \) in lambda
  \( A_2 = \{ p \mid p \) is a nearest point to \( g \) on Boundary(\( O_2 \))\}\)
Choose one point \( p \) from \( A_2 \)
\( g \)-bar is the intersection point between the half line
  from \( p \) to \( g \) and an edge in \( E \)

Return the path from \( s \) to \( s \)-bar in straight line movement
Use Dijkstra's shortest path search to find a path from \( s \)-bar to \( g \)-bar[29]

---

**Figure 22.** Path Generation by Global Planner
Meng's system also has the capability of avoiding obstacles. If an obstacle is detected blocking one of the paths, Meng's algorithm simply removes the line segment in the Voronoi diagram that intersects the obstacle and plans a new path using the remaining segments.

3.3. Details of the Constraint Analyzer

The next level of the hierarchy we wish to discuss is the constraint analyzer level. As in the previous section, we would like to develop some criteria for determining what representations and techniques are appropriate for this level. In addition, we will determine what, if any, classical representations and algorithms fit within this category, and we will attempt to compare them using the criteria developed. Finally, we will detail an example analyzer that we used to physically experiment with this hierarchy.

3.3.1. General description

The constraint analyzers are responsible for verifying some aspect of the proposed global path. Since each analyzer is very specialized, they each may make use of very different information in their analysis. Some may use the global map for information, some may use information from the local planner. Their basic goal is to catch un navigable paths before they are passed to the local planner. Whenever an analyzer encounters a problem with a path, it needs to weigh several alternatives it has for dealing with the problem. First, if the problem is very minor, such as if the robot follows the proposed path, it will pass closer than safety tolerances to an obstacle, it may still pass the path on to the local planner with the assumption that that planner will take care of the problem. In a more serious situation, such as a definite collision with an
obstacle, the analyzer can either attempt to deal with the problem itself, or it can return the path to the global planner. Since problems of this type are usually localized, it is more efficient for the analyzer to deal with it. Furthermore, signaling the global planner that the global path is unnavigable may not solve the problem if the global planner does not correctly handle the obstruction. It is generally best to solve path problems at this level when possible. The analysis level of the hierarchy should pass on the path within a reasonable amount of time, say a minute maximum. Most of the time, though, the response time of this layer should be only a few seconds.

3.3.2. Types of Representations and Algorithms

Since other researchers have not adopted the constraint level concept, there are very few planners specifically designed for this purpose. However, many path planners can be easily modified for this purpose. There are three kinds of analyzers that are significant to almost any work environment: the first of these enhances the model of the robot to detect collisions more accurately, the second adds understanding of "unusual" obstacles, and the third adds rotational capabilities to the path planning. Using a polygonal representation for the robot is usually sufficient to detect most collisions in normal workspaces. However, more complicated three-dimensional techniques, such as solid modeling, can be used if it becomes critical. Three-dimensional models could also be used to provide graphical simulations of the motion.

The representations for unusual obstacles would vary depending on the particular obstacle. For example, overhead pipes might be modeled by storing the height above ground, diameter, and endpoints of each pipe. This information about each kind of unusual obstacle is stored in its own obstacle map. The given global path would be tested for collisions using this map. We can also add temporal constraints, such as doors closing at a particular time or schedules. In this case, the analyzer would use the
acceleration, velocity and distances to estimate the time-of-arrival for particular waypoints. It could then compare this to a schedule.

Most of the global planners that currently exist cannot handle rotational motions, and the systems that do exist usually assume a robot with three independent degrees of freedom. If the robot is fully omnidirectional, then we can use the techniques developed by Brooks, Lozano-Perez and others to handle this case[32]. These techniques usually convert the robot and the obstacles to C-space. By solving the planning problem in this space, rotation is encoded as another degree of freedom identical to the translational ones. We mentioned before that C-space involves a large amount of calculation, thus minimizing the subspace that the planner has to analyze is critical to the response time. Since we already have an analyzer that detects the location of each collision, we pass the rotational analyzer only the region nearby the collision. In this way, the rotational planner only deals with a few obstacles. Unfortunately, most mobile robots are not omnidirectional; in this case, we need a rotational planner which understands these robots. Jacobs et al. present a technique that assumes a robot with a finite diameter of rotation[34].

![Figure 23. Encoding of Smooth Trajectories](image)
Their technique uses paths consisting of line segments and tangential arcs. They show that since the trajectories in this scheme can be uniquely parameterized, they can form a configuration space for all possible paths of this type. If it is not crucial to generate an optimal path, they show that the computations can be greatly reduced by using a grid based approach for searching for valid paths. A technique such as this could be used for generating local rotations to avoid obstacles.

3.3.3. Example Constraint Analyzer

We chose to experiment with two simple constraint analyzers; the first one checks the global path given to it for collisions based upon a polygonal representation of the robot. This analyzer uses the polygonal representation of the robot and the polygonal map used by the global planner to verify the path. To test the global path, it simply navigates the path using the robots' representation and checks for collisions between the polygon of the robot with the polygons of the environment. If a collision occurs, the analyzer signals the global planner that the path is un navigable. If no collision occurs, then the constraint analyzer passes the path to the local planner which can then execute it.

![Image of a robot arm and an overhead obstacle]

*Figure 24. Detection of Collision with Overhead Obstacle*
The second analyzer we created handles low ceilings and overhead obstructions. It models the obstruction as a two dimensional polygon placed at some altitude. We assume that the robot may be carrying some object that would change its overall height; presumably, this object could be removed or repositioned if it is too high (see figure 24). This analyzer similarly traverses the path with a model of the height of the robot; and if this model intersects the overhead obstacle, it signals the global planner that a collision has been detected.

3.3.4. Multiple Constraint Analyzers

If we add more than one constraint analyzer to the hierarchy, we need to add conflict resolution in case these analyzers generate two or more different paths; we can handle this with the following organization.

![Diagram of Multiple Constraint Analyzers]

*Figure 25. Control of Multiple Constraint Analyzers*
If the local planner is passed more than one valid path, it could simply choose to navigate the first path it is given, or it could choose based upon some criteria. We might also add a mechanism so that only the analyzers that are appropriate to a given situation are "awake"; this will limit wasted computations. Only those paths that satisfy all of the constraints should be passed on to the local planner.

3.4. Details of the Local Planner

In this section, we will detail the role of the local planners and determine some criteria for planners in this category. In addition, we will determine what classical representations and algorithms fit within this category, and we will attempt to compare them using the criteria developed. Finally, we will detail the local planner that we implemented to physically experiment with this hierarchy.

3.4.1. General Description

The local planner handles navigation and obstacle avoidance using the available sensors. Its work environment extends approximately to the range of the sensors. There are several different kinds of sensors that can be used individually or together for the planner; they include ultrasonics, vision, LIDAR, and force sensors. Since the robot may be moving at fairly high rate (a couple of feet per second and higher), the response time at this level has to be very fast. However, it is important to differentiate between the time acceptable for calculating the local path, which may be on the order of a few seconds, versus the response time if an obstacle is detected, which has to be much less than a second. A good local planner should be able to navigate around small obstacles without requiring a new global path, but if that is not possible, it may request a new path.
3.4.2. Types of Representations and Algorithms

The representations used by the planners tend to be custom tailored to the particular sensor; and as such the performance of the algorithms depends heavily on the performance of the sensor. We will look at the representations related to ultrasonics and LIDAR sensors.

Ultrasonic sensors provide a fairly accurate range estimate if they are directly facing the obstacle; otherwise, the range accuracy varies depending on the angle of incidence. Furthermore, the sensory cone is approximately 30 degrees[69]. This is a very large cone; in consequence, very little resolution is possible. Usually, Ultrasonic sensors are placed around the robot so that they measure the distance in several directions at once. This produces a wide-angle range profile at a specific height. One technique used for Ultrasonic navigation stores the vertical surfaces of the obstacles the robot might encounter. When the ultrasonics scan the environment, the system builds up a model of the vertical surfaces of the obstacles[72]. This representation is sufficient for simple local navigation. However, it is difficult to match this representation with any global map representation; thus it would be hard to use this in our hierarchy. Unfortunately, most ultrasonic-based algorithms have similar properties. In general, the ultrasonics are too coarse to provide any detailed information about the environment. They function best as a blunder device to avoid immediate collisions, not to provide information about the environment.

One technique that has been used to improve the accuracy of ultrasonics is to place several such sensors in an array and use phased-array techniques to narrow the beam. This does produce a narrower beam, and thus greater resolution in the overall scan. The difficulty is in combining the array (which is usually very large) with an accurate scanning mechanism. Furthermore, such a system would still have problems
with the angle of incidence to the obstacle. Techniques of representation used by these scanners are very similar to those used for LIDAR sensors.

LIDAR is a very good sensor for providing local obstacle information. The normal configuration is a down-looking view of the environment. From this configuration, the most common technique is to convert the range information into a top-down (i.e. map) view. Notice that the scanner samples discrete points of the scan; because of this there is a gap between the sampling points which increases with the distance. The conversion process to create the top-down view involves converting the range points into horizontal displacement from the robot, and then interpolating between the points to generate the obstacles. Furthermore, regions of the generated map will be obstructed; these regions are called occluded regions. Usually the path planner assumes that these are filled with an obstacle until it can verify what they actually do contain[31]. Once this top-down view is generated, various different general techniques for path planning (such as quadtrees, etc.) can be used for finding a local path. LIDAR scanners provide the most information about the environment in a reasonable amount of time of any of the common sensors.

3.4.3. Example Local Planner

The local planner we implemented uses ultrasonic sensors to determine obstacles. The system as implemented is fairly simple; it can detect obstacles in the path and make slight adjustments to the path. The planner uses a set of ultrasonic sensors located around the robot as shown in figure 26. With the given arrangement of sensors, we can determine an accurate distance to obstacles directly ahead of the robot, and an approximate distance to those on the side.
When an obstacle appears in front of the robot, we can determine the following information:

1. If the range to the obstacle is the same in both A and B sensors, it is a large obstacle that intersects both cones.
2. If only sensor A detects the obstacle, then somewhere within A's cone is the edge of the obstacle. Similarly if only sensor B detects it.

![Obstacle (Within single cone)](image)

*Figure 28. Small Obstacle Detection*

Whenever an obstacle gets within 100 mm of the robot, an emergency stop signal is sent to the base and the local planner signals that it cannot execute the path. Otherwise the local path is executed as given. Obstacles on the side of the robot are ignored unless the robot comes within 100 mm of them. However, if the robot is moving at a shallow angle to the obstruction, a slight course correction may be tolerable. In this case, the robot turns just enough to stop approaching the obstacle. This may cause a growing error with the requested path; when the error grows too large (i.e. over 100 mm) the local planner signals that it cannot complete the path. If the error is only minor, than the course can be corrected in the next segment of the global path. Although this local planner provides only minor path planning capabilities, it was sufficiently advanced to avoid collisions and
provide experimental results with the other levels of the hierarchy. The results it fed back to the higher level planners did provide them with rough information about obstacles. In general, the speed and simplicity of this planner give it a very fast response time, and thus it would work well in conjunction with a larger, more complicated local planner that provided more detailed obstacle information.
Chapter 4

Distributed Blackboard Architecture

4.1. Design Decisions

There are a number of decisions regarding the overall structure and organization of the blackboard that we made. These decisions reflected the particular application of the blackboard to hierarchical path planning. Further constraints on the system were dictated by the hardware environment and our overall design goals. Some of these constraints were as follows:

1. Heterogeneous computer and networking environment
2. Need to pass paths around
3. Need to access blackboard from any process
4. Software needs to be easily transportable

We decided that the issues and techniques involved in distributed databases provided a useful framework for beginning our design. As we have seen, other blackboard researchers have also used techniques from distributed databases, especially its transaction model. However, we also realized that distributed databases can get very complicated in maintaining the consistency of the overall database. Significant overhead in code and access time is needed by the locking and data consistency systems. For that reason, we will find it much more useful to develop a "simplified" transaction model
which not only significantly decreases the complications of distributing the data, but also is much more appropriate to the transactions in a blackboard system. In this section, we will develop this simplified transaction model, and then compare its speed and complexity to the regular database model.

4.1.1. Uniform Blackboard Interface

We recognized that in our robot environment most of the Knowledge Sources are actually large, independent processes. It is important that the blackboard is accessible to all processes at any time. Thus it does not make sense to create a standalone program with its own user interface, debugger, etc. Instead, we want the blackboard to run as a background process which constantly monitors for new connections. In this system, the Knowledge Sources handle the screen interface, debugging, etc. and the blackboard daemon handles only the object distribution and process invocation. And as we previously mentioned, there should be no apparent difference between data transfers on a single processor versus data transfers between processors.

4.1.2. Simplicity of blackboard

One of the greatest difficulties in the robot environment is the heterogeneity of computers and connections. In order to make the distributed blackboard system easily portable to many different Operating systems, programming languages, etc., we had to remove most of the complexity from the blackboard itself and instead place these tasks on the Knowledge Sources. Thus, for example we opted for a data-driven scheduling system. This separated the task of scheduling from the blackboard itself, and instead placed it on the Knowledge Source level. For example, if we had several different Knowledge Sources that were capable of processing the same event, we would create a separate scheduling Knowledge Source to determine which of these KS should be
invoked. By handling the scheduling this way, it makes it easy to modify the scheduling algorithm, while significantly reducing the complexity of the blackboard software. This technique functions well as long as the number of competing hypotheses is reasonably low, which is true in our particular application.

The main philosophy in this blackboard system is to provide the basic data distribution and automatic routing without incurring the overhead inherent in distributed databases, nor the added complexity of the schedulers, etc. Since the Knowledge sources, unlike the blackboard itself, will be implemented and run only on a single kind of machine, we don't have to worry as much about minimizing their complexity.

4.1.3. Transparency

As we have seen, there are advantages to having a completely transparent system in which local and global calls are handled uniformly. We decided that making the system transparent provided a large degree of design flexibility that we wished to use. Thus all accesses to the blackboard are uniform regardless of where the datum is created. Most distributed databases are also transparent, as well as some distributed operating systems. However we should remember that transparency is usually achieved at the cost of complex code and large overhead. The most efficient distributed systems with transparency handle local transfers separately from global transfers in the underlying implementation. This enables the systems to maintain a higher transfer rate between processes on the same machine than on separate machines, while still maintaining the appearance of a uniform environment.

4.1.4. Objects as Datums

In the background section, we showed that interaction between Knowledge Sources and the blackboard consisted of accesses to structures of data. These structures
define the *atomic* level of interaction with the blackboard. Any access to the blackboard consists of accesses of whole structures, and any data consistency or fault tolerance of the system is guaranteed on the structure level. In our formalization, this structure of data is called an *object*. These objects are not the same as object-oriented objects because they do not include any code. An object roughly corresponds to a structure in C. Each object has fields of various sizes and data types depending on its predefined structure. For example, we might have an object which represents a global path. It would include the following information.

<table>
<thead>
<tr>
<th>Flags</th>
<th>Special flags indicating errors, conditions, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumSeg</td>
<td>Number of path segments in this path</td>
</tr>
<tr>
<td>Wayptx[NumSeg]</td>
<td>The X coordinate of all of the waypoints of the path</td>
</tr>
<tr>
<td>Waypty[NumSeg]</td>
<td>The Y coordinate of all of the waypoints of the path</td>
</tr>
</tbody>
</table>

More complicated objects might include references to other objects to create linked lists, etc. However, in our implementation we did not include this feature. When we discuss a particular kind of object without actually associating any data with it, we are talking about the object *type*. If, on the other hand, we discuss an object with some particular data written to it, we are talking about a particular *instance* of that object. This follows the common distinction between types of data and instances thereof.

4.1.5. Object Transaction Model

We wish to develop a transaction model for our distributed blackboard that parallels those in distributed databases. However, accesses to data in blackboards are distinctly different from those in databases. Specifically, there is an important distinction in blackboards between reading and writing data. When a Knowledge Source writes
information to an object, it is handled similarly to writing data to a database. The transaction model is given by:

1. The transaction is begun
2. Data is written to the object
3. The transaction is committed or aborted

The calling process is only blocked temporarily when it begins the transaction if another process is currently writing to the same data. This is identical to the database model. But reads have a distinctly different model. If one looks closely at the basic organization of blackboards, the system is structured such that a read on an object will block until valid data for that object becomes available. Thus a simple model for this read transaction would look like the following:

1. The transaction is begun
2. A read on an object blocks until valid data is available
3. The transaction is committed or aborted

Notice that if a process is blocked on a read, at least one write by another process on that object must be allowed to complete, otherwise a deadlock will occur. A significant improvement in performance can be made on this model by converting the read to an interrupt driven system. The transaction model for this system would be:

1. A process associates an object with a procedure
2. The process continues...
3. When the object becomes available for reading
   the procedure is invoked
4. The transaction is begun (no blocking)
5. data is read from the object
6. The transaction is committed
In this model, an object becomes available for reading when at least one other process writes data to it. This transaction is an infinite loop; whenever another process writes to the same object, the procedure is again invoked. Notice that the procedure receives an instance of the object; each time the procedure is invoked (i.e. each time data becomes available), the instance may or may not be the same instance as the previous time it was invoked. Another significant variation from database transactions can be made on the write sequence if one can make the assumption that writes in separate processes always occur to distinct instances of objects. In other words, instances of objects created in one process are never modified by other processes. This seems, at first, to oversimplify the transactional model. However, if we look at the conditions under which an instance of an object might be modified, we see that most such situations can be easily handled even with this assumption, and those that don’t can be rewritten to work within this framework.

In the first situation, a process creates an instance of an object, writes data to it, and commits the transaction. Later he might wish to write further data to this instance; this is completely allowed since he created the instance to begin with. In a more common situation, a process registers an interest in an object. When the process is invoked, it reads data from the object, writes some data to it, and then commits. If we assume that the instance this process receives is only a copy of the instance that the original process created, then our assumption is valid. This does not exactly follow the "pure" blackboard architecture; in the pure system the second write would cause the data seen by the first process to also be modified. In our system the data in the original object will not be modified since it is actually a separate instance. If a process can expect that an object it has created will be modified by another process, it should register an interest in that object and handle its local writes and the foreign modifications separately. When it wishes to make its changes to the object available, it would explicitly distribute the
object. This does not remove the ability to share data. Instead, it forces the Knowledge Sources to clearly define when they wish to make their changes available to other Knowledge Sources.

4.1.6. Database Transactions versus Object Transactions

Having developed a simple transaction model for the system, it would be good to compare and contrast it with the normal database transaction model. As we mentioned in the background, insuring data consistency during concurrent accesses to the same data object in a distributed database implies that databases need complicated locking mechanisms. This becomes even more complicated if the database maintains caches of data on each of the machines. In this case it becomes a cache coherency problem.

In our transaction model, Reads and Writes by different processes cannot simultaneously occur to the same object instance. Thus we don't have to worry about the data integrity when Reads and Writes are interwoven. This, in turn, implies that we don't have to worry about implementing a locking mechanism for the shared data objects across multiple machines. Whenever a routine wishes to start a new transaction that writes to an object, it just locally creates the object. When the write transaction completes, the object is distributed to all other machines that have processes which have expressed an interest in it. It is possible to make an extension to this system that allows concurrent writes to a single instance of an object among processes on the same physical machine by using the operating system and hardware support for shared memory. This enhancement can be made such that it only slightly increases the overhead of other transactions.
4.2. Program Interface and Organization

In this section we will outline the program interface to the blackboard. In Unix systems, the blackboard itself (i.e. the daemon) is an independent process. User programs interface to the blackboard through a "stub" code which communicates with the blackboard daemon and contains the basic commands. Objects of data consist of pre-defined structures that all processes can (theoretically) access regardless of what processor they are on. Objects are distributed locally and over the network to whatever Knowledge Sources require the information. We will discuss the role of the daemons, the user-interface routines, and the process by which an object's structure is defined.

4.2.1. Daemons

As has already been stated, the blackboard itself consists of several daemons, up to one per processor. Daemons have many tasks; fundamentally they maintain a map of the connections with other daemons and with the Knowledge Sources. It is important to realize that there is no central controller for the communications; all communications and control is distributed among the daemons. This reduces the bottleneck of communications. Daemons constantly monitors for three different kinds of communications: connect messages from other daemons, connects from Knowledge Sources, and actual object distribution commands. When it receives a connect message, it records information on what kind of process is connecting and its address to the map of connections. This information is used when an actual command is received to distribute data or commands. Daemons maintain additional information about the kinds of objects the local Knowledge Sources have an interest in, as well as what objects the other daemons are interested in.
4.2.2. Accessing and Distributing Objects

When a process wishes to write data to a new object, it must first create the object. The local daemon allocates space for it and records various information, including pointers, its size, type object type, etc.:

\[
\text{handle\_to\_obj} = \text{CreateObject(objecttype, objectsize)}
\]

The objecttype is a number from a "common list" which all processors & all daemons have.

Notice that this routine returns a handle to the object, rather than a pointer. Handles are a pointer to blocks of information about the object. These information blocks contain additional information that the daemon and other routines need to manipulate the object. Knowledge Sources do not usually reference this information block directly. Handles make it possible to provide a uniform program interface even though the underlying structures and object storage techniques are extremely different. To retrieve the actual pointer, one calls the following routine:

\[
\text{obj\_pointer} = \text{AccessObject(handle\_to\_obj)}
\]

This pointer is then used to read and write data to the object. Only this particular Knowledge Source has access to the data, thus conflicts do not occur with other Knowledge Sources trying to access the same data. Once the process is ready to commit the transaction, it calls the following routine:

\[
\text{err} = \text{DistributeObject(handle\_to\_obj, objectsize)}
\]
Where the handle_to_obj is the same handle the process was passed in CreateObject. This routine returns an error if the object, for some reason, cannot be distributed or an error occurs.

4.2.3. Registering Interest

If a process wishes to access an object created by another process, it has to register an interest in that kind of object. It does this by calling the following routine:

\[
\text{err = RegisterInterest(objecttype, myprocedure)}
\]

In this case, myprocedure is a procedure to be called when an instance of the object of type objecttype becomes available. The procedure has the form:

```c
myprocedure (handle_to_obj, object_size)
handle handle_to_obj;
int object_size;
```

The procedure is passed the handle to the object as well as the size of the object. In order to access the data, this procedure calls AccessObject() passing it the handle_to_obj that it received. Notice that in this case the procedure does not call CreateObject since it is passed the handle. The object that the process receives is a copy of the object originally distributed; thus it is a different instance. The process is allowed to read and write data to the object. If the process wishes to share the modifications with other Knowledge Sources, it simply calls DistributeObject() with this handle.
4.2.4. Defining an Object

There are two parts to defining an object. First, as we have already mentioned an object has some structure. Its structure consists of a number of fields which may contain strings, numbers, etc. These fields can contain almost anything that a regular C-language structure could contain. The only exception to this is pointers; since pointers are valid only within a particular process, they lose their meaning when sent to another processor. Instead, objects should only reference other objects. Thus handles must be used to refer to other objects. Once the objects' structure is defined, a unique identification number is associated with it. This number must be the same on all machines that are running the blackboard. Furthermore, the structure must be (logically) the same on all machines. Each machine has a file containing the unique id numbers for the objects, their names, and the structure definitions. These files are used whenever the stub code is compiled into the user code; thus object structures cannot be changed dynamically. If the objects are being distributed across completely different operating systems, such as from the Unix processor to the Explorer, it is likely that there will also need to be some conversion routines. All objects are distributed using a common byte ordering. When an object is received on a machine that uses a different ordering internally, a conversion routine is used to convert the object to this format before it is given to the local processes. Depending on the extent of the changes necessary, this conversion will increase the overhead of the blackboard. Data transferred over the network uses the standard bytes as defined by the underlying protocols[71].
4.3. Implementation of Transparency

The transparency of the system involves a complicated interaction between the daemons and the Knowledge Sources. This section details how this transparency is implemented.

4.3.1. Communications Structure

The daemons have many roles in the overall communications. In this implementation, all global communications uses Sockets. Sockets are a software mechanism for providing reliable inter-process communications between two process regardless of what machines they are running on. Local communications can be implemented in several different ways: it can used shared memory; it can use special communications channels; or it can also use Sockets. Daemons store information on both the interconnection network and the objects being distributed.

Figure 29. Communications Level of Blackboard
PLEASE NOTE

Page(s) not included with original material and unavailable from author or university. Filmed as received.

83-84

University Microfilms International
4.3.6. Inter-Daemon Communications

The daemons send the following kinds of messages among themselves:

1. Register Forward Interest
2. UnRegister Forward
3. Forward Object

Whenever a Knowledge source registers an interest in a type of object, the daemon sends a message to all of the other daemons indicating that it wishes to receive objects of that type. Thus it *logically* broadcasts this message to all daemons, although *physically* it sends the message to each daemon in its active list one at a time. In this way, each daemon knows exactly which other daemons have requested the object. Thus when an object needs to be distributed, a daemon looks in both its local table as well as its daemon table to determine where to send it. A picture of the entire process of registering interest and forwarding objects is given in figure 30. Whenever a daemon process dies, the other daemons remove any requests for objects from that daemon from their lists. Furthermore, if all of the Knowledge Sources with interest in a kind of object either die or remove their interest, than the local daemon no longer needs that object from the other daemons. So it sends a message to the other daemons un-requests that object. One of the advantages of this architecture is that local distribution is handled separately from global.
4.4. Assumptions and Simplifications in our Design

The distributed blackboard system that we have implemented here was designed specifically for the environment and constraints of the robot, as well as being specifically applicable to hierarchical path planning. Because of these and other reasons, we have made simplifications and assumptions in relation to "general" blackboard theories. These need to be thoroughly understood to properly determine the usefulness and flexibility of the system.

4.4.1. Timestamps

Timestamps are used in many blackboard systems, especially when it is critical to fuse data together or combine competing hypothesis. But on a distributed blackboard
system, they create difficulties. In a distributed environment, trying to ascertain an "absolute" time among various machines is difficult, because it is almost impossible to synchronize the separate clocks of the processors; one processor would have to be designated the "true" time. Thus whenever a timestamp is required, a message would need to be sent to this processor requesting a timestamp. If every object needed a timestamp when it was distributed, then this would form a severe bottleneck.

If we look at the particular application of hierarchical path planning, we can make a very important simplification: for most applications, the Knowledge Sources only need timestamps that are locally generated. When we need a timestamp for these objects, it does not have to be global to the whole distributed blackboard; rather, it only has to be uniform on a single processor. By making this simplification, we can reduce most timestamp look-ups to a local procedure call. We can still have the global timestamp available, but it should be used only in unusual circumstances when a truly absolute time is needed. If we can determine a means by which we can measure the difference in the clocks, we can create a translation from the local to global time. The timestamp information is recorded with the object so that it is always available for all Knowledge Sources.

4.4.2. Errors and faults

The system that we have designed does not have a large amount of fault detection or correction in the blackboard itself. Instead, we depend upon three factors for insuring data consistency:

1. Underlying network correction

Since we are making use of the error correction in the underlying network software (i.e. Sockets, etc.), these provide us with guaranteed communications. Thus we
don't have to worry about the packet level errors. Since the throughput of this application is relatively low, we can afford to leave the low-level networking protocols in place. If we wished, in the future, to replace that software with our own, we would have to deal with this in greater detail.

2. Knowledge-source level acknowledgement

We also placed a simplistic acknowledgement system in the object map for the hierarchical path planning system. This provides a simple high-level detection of errors. This will be explained in more detail in section 5.

3. No broadcast messages

We also lowered the possibility of error by designing the communications plan such that at the physical level no broadcast messages are used for inter-daemon communications. Instead, each processor communicates (at one time) with a single other processor. In most underlying network software only one computer is guaranteed to receive a broadcast message; all the others may or may not[71]. Thus broadcast messages are not reliable. In our system, the processors can detect errors in all communications using the underlying software.

The other significant assumption we made was that we would not need the system to recover after a major fault. Since the state of the system includes the state of the physical robot, it is impossible to completely recover after a system fault anyway. Thus if the system crashes, it must completely restart.
4.4.3. History

It was also clear that in the application of hierarchical path planning, we did not have to retain "old" information. Whenever a new knowledge source starts up, it will receive any objects that are distributed after its startup; it cannot access information that was previously distributed. However in our application, path planners and path executors (which constitute the majority of the Knowledge sources) want the most up-to-date information anyway. Thus this does not fundamentally limit the system. The gain we achieve through the reduction in the complexity of the transaction model more than justified any perceived loss.
Chapter 5

Advanced Path Planning Using Blackboards

5.1. Fusion of Path Planning and the Blackboard

We now need to combine the path planning hierarchy with the distributed blackboard system presented above. Each of the pieces of the planner are implemented as processes; these processes communicate by accessing the objects of the blackboard. There are several issues involved with combining the hierarchy with the blackboard. First, we have to make sure that the throughput of the blackboard is sufficient to cope with the data rates required by the planners. Second, we have to design the object-level interface between the planners and the blackboard. And third, we need to describe the interactions between planners based upon the objects received. While we are doing all of those things, we need to take into account where the planners reside, and which links between processes are "fast" and which are "slow".

5.1.1. Throughput and Speed Requirements

Before we combine the planner and blackboard, we need to get an idea of the throughput requirements that the planner needs to make sure that the blackboard system can handle the data. In the planning system as we have described it, there are three kinds of data passing between the levels of the hierarchy: paths, maps and commands. Paths are
transmitted from the global planner to the analyzers, and from the analyzers to the local planner. Each path consists of a series of waypoints, which are simply two numbers representing the next X and Y position of the robot relative to its current location. If we refer to figure 22, we see that such a path has a maximum of about 25 waypoints. Returning to our model environment, we see that a global path may pass through several rooms. Assuming that our global environment includes about 50 rooms, we can pessimistically assume that a worst-case path will pass through all of those rooms. Thus a maximum path in our environment would entail 1250 waypoints. Since each waypoint includes 8 bytes of information, this totals to 10000 bytes of information. Furthermore, the global path is generated not more than once a minute. This path, with possible modifications, is also passed from the analyzers to the local planner.

The constraint analyzers can also request the global map object; this only occurs when the system is initialized, when the robot enters a new domain (such as going from inside a building to outdoors), or when significant changes have occurred to the environment. Thus this object might be distributed once every five minutes. This object consists of multiple polygons. Each polygon is represented as a set of X,Y points similar to the waypoints above. If we assume an average object has six vertices, this would be 24 bytes per obstacle. Based upon the workspace model, we can estimate an average room to contain 20 obstacles which would be modeled in the global map. And with 50 rooms, this gives a total of 1000 obstacles in the workspace. Thus a map would consist of approximately 24000 bytes sent every 5 minutes.

The last category of objects being distributed are commands. These are each very small, but they need to pass quickly through the system. If we assume an average command contains 10 bytes of information, and we need 100 messages per second this implies 1000 bytes per second for commands. If we total this with the other numbers we see that the blackboard needs to handle roughly 2K bytes per second, or 16Kbps. We
have measured an effective throughput of the system at roughly 111Kbps. Thus we see that the blackboard is more than adequate at handling all of the transfers required for the planner.

5.1.2. Defining Global, Local, and Feedback Objects

Now that we know the general interactions of the different levels of the planning hierarchy, we need to specifically define the objects that are created and received by each planner. This creates a data-based control structure for the hierarchy. If we begin with the global planner, we see that it responds to the following events:

1. When the global planner is given a goal point, it creates a path from the current robot location (which it is given by the low-level routines) to the goal.

2. If a previous global path it has generated is impassable, it needs to create a new global path.

3. Another planner may request a copy of the global map.

All of these events can be serviced using the following set of objects:

<table>
<thead>
<tr>
<th>Output Objects</th>
<th>Input Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLOBALL_PATH</td>
<td>GLOBALL_GOAL</td>
</tr>
<tr>
<td>GLOBAL_MAP</td>
<td>CURRENT_LOCATION</td>
</tr>
</tbody>
</table>

- path consisting, normally, of a series of waypoints
- map of the global environment
- the goal position for the robot motion.
- current robot location
Next we look at the constraint analyzers. We see that they react to the following situations:

1. The analyzers need to know whenever a global path is created. When they acquire this path, they will analyze it. If it is totally impassable (even with minor modifications), it must reject it. Otherwise, it creates a local path.

2. If a previously generated local path is impassable, it needs to create a new local path. If it cannot, it needs to have a new global path generated.

The analyzers use the following objects:

**Output Objects**
- LOCAL_PATH - the local path
- OBSTR_GLOBAL_PATH - this is created if the given global path cannot fit the constraints

**Input Objects**
- GLOBAL_PATH - the global path
- OBSTR_LOCAL_PATH - the local path was impassable

Finally, the local planner handles the following situations:

1. Given a local path, it attempts to navigate it. It may make minor modifications to the path to avoid small obstacles.

2. If a local path is unnavigable, it must reject it.

3. When it completes a local path, it should inform other planners of its final position.
These can simply be handled by:

**Output Objects**
- OBSTR_LOCAL_PATH - when it fails to execute the local path
- CURRENT_POSITION - when it finishes the local path

**Input Objects**
- LOCAL_PATH - the local path

All of these objects together form an object map of the hierarchy, and this map describes the control organization. If we create a picture with the objects and their dependencies we generate a picture like the following:

![Diagram](image)

**Figure 31. Object Dependencies**

This picture shows how path problems in the lower level planners feed back to the higher level, and thus modify the higher-level paths.
5.1.3. Conflict Resolution with Multiple Constraint Analyzers

We originally stated that there were going to be several constraint analyzers functioning simultaneously. In order to handle this situation, we showed in section 3.3.4 that we needed a technique for a) determining which analyzers are appropriate to the given situation, and b) if there are several paths which satisfy all of the constraints, which one should be passed to the local planner. Using the blackboard system, we can handle this by creating two additional knowledge sources. One knowledge source is responsible for determining which analyzers are appropriate; it draws upon a list of categories of which one or more may be valid. For example, we might have an outdoors category, an indoors category, a hallway category, an office category, etc. The knowledge source would use the \texttt{GLOBAL\_MAP} and \texttt{CURRENT\_LOCATION} objects to determine which categories are valid for the domain. Then it would create a \texttt{CURRENT\_CATEGORIES} object. Since each of the analyzers have the best knowledge about what they are good at, each of the analyzers would read the \texttt{CURRENT\_CATEGORIES} object and determine whether they are appropriate to that situation. Whenever they are appropriate, they register an interest in the \texttt{GLOBAL\_PATH}. Otherwise they unregister interest (if they have previously registered it). The other knowledge source we need would select between the various \texttt{LOCAL\_PATH}s available from the analyzers and choose a single one to pass to the local planner. Thus it would register an interest in \texttt{LOCAL\_PATH} objects, and generate a single \texttt{E\_LOCAL\_PATH}. There are several different criteria this KS could use to select between the paths. The simplest technique would pick the first available path and discard all remaining paths. A better criteria could use some knowledge of the environment to determine which path is most likely to succeed. Or the KS could use information on the algorithm and representation used to generate the path to make a judgment. In our experimental system, we implemented this knowledge source using the first-path criteria. Notice that the test system only includes two
constraint analyzers; the abilities of these conflict knowledge sources would be much more significant if we had many analyzers. We did not implement the categories knowledge source.

5.1.4. Comments on Interface

We can make some general comments on using data-driven scheduling versus having a scheduler for the application of path planning. First recall that the original reason we went with a data-driven scheduler was to avoid the added complexity in the blackboard needed by a built-in scheduler. In the case where we have only a single global, analyzer and local planners, it is obvious that there will not be competing hypothesis at any instant, and thus having a data-driven scheduler does not cause problems. In the more complicated situation with multiple analyzers, the scheduler is needed to a) to determine which analyzers should be activated, and b) to decide which path should be passed to the local planner. But as we saw in section 5.1.3, we can easily modify the objects and add knowledge sources to handle these problems. Furthermore, built-in schedulers tend to follow relatively "dumb" techniques for determining which hypothesis to choose (an exception to this is the meta-blackboard for scheduling in BB1[4]), whereas the knowledge sources we introduced in 5.1.3 include much more knowledge about which analyzers are appropriate, and which paths are most likely to succeed. Thus having data driven scheduling is much more flexible.

There were several reasons why the path planning hierarchy interfaced to the blackboard easily. The first factor in this was the clearly defined structure of the hierarchy. This structure defined how to subdivide the problem, and the role of each knowledge source. We decided that each planner of the hierarchy would correspond to one knowledge source. Given this, it was not difficult to create the associated objects for communications. There are two factors that could have made the interface much more
difficult. The hierarchy we implemented has a reasonably simple command/response system. Each path generated could be considered a command, and the response from the receiver of this path is either success or an obstructed path. If this command/response protocol included a significantly more complex interaction between knowledge sources, using objects for passing messages might cause too many delays and would make it more difficult to implement. Also, if the overhead associated with the blackboard was significantly higher, it would have made it much more difficult to get the hierarchy to produce a navigable path within the time constraints. On the positive side, having the blackboard handle all communications significantly decreased the time developing communications software. In the previous systems we used, all communications were point-to-point using custom software on both sides. With the blackboard, all data is automatically available on all machines, and the system can be quickly and easily reconfigured.
Chapter 6

Experimental Results

6.1. Performance Monitoring and Debugging Issues

In order to monitor the performance of various pieces of the blackboard, we created a knowledge source specifically to monitor the objects being distributed, and to generate arbitrary objects for testing different pieces of the system. The developer can interactively control the kinds of objects the debugger is generating and the frequency at which they are distributed. It generates displays of the information contained in the objects it receives, and this information can be stored in a file or displayed on the screen. We used this program first in debugging the message passing of the blackboard, then in debugging each of the layers of the planner, and finally in measuring the performance of the blackboard and the planners. This Debugging Knowledge Source we created uses direct calls to the UNIX time subroutine to generate timing information.

While debugging, we needed to monitor all of the data being sent through the blackboard. While the blackboard itself was being constructed, we needed to know the exact number of bytes sent in each part of the message. If any miscount occurred on either the sending or receiving side of the message, the receiver either locked waiting for nonexistent data, or it misaligned the following message. This happened frequently between the UNIX processors and the LISP processor, because of the byte ordering and
conversions required between them. When we implemented the hierarchy using the blackboard, the monitor enabled us to independently verify each module as it was installed. For example, when we tested the constraint analyzer, we configured the debugger to monitor for LOCAL_PATHs and OBSTRUCTED_GLOBAL_PATHs. Then we had it create a GLOBAL_PATH. Given this, the analyzer produced a LOCAL_PATH, which was displayed on the screen. The debugger can also record the object information to disk, including the timestamp information. In this way, a permanent record can be kept of the order and contents of all objects distributed on the blackboard.

6.2. Demonstration of Complete Hierarchy

In order to validate the complete hierarchical system, we ran a demonstration with the Rice-obot navigating in a cluttered environment. The environment we used was the world room of the Cooperative Intelligent Mobile Robotics (CIMR) laboratory. We set up a set of boxes which looked like the following:

![Figure 32. Test Environment](image-url)
The Rice-obot began at the start location in the picture on the right and was commanded (by receiving a GLOBAL_GOAL object) to move to the goal point. We wanted the demonstration to show the feedback and planning involved in all three levels of the hierarchy. The basic global planner is demonstrated by planning the global path. For demonstrating the constraint analyzer, we placed the obstacles such that obstacles (A) and (B) were too close together for the robot to fit between, yet far enough apart such that the global planner would find a path between them. In this circumstance, the constraint analyzer will catch the problem before passing the path to the local planner, and instead send a constrained global path back to the global planner. We placed block (C) such that the robot would pass very close to the obstacle while traversing the path. In this circumstance, we wanted to verify that the local planner would make a slight coarse correction to safely pass the object.

![Diagram](image_url)

Figure 33. Motion Correction for Obstacle

In this way, the local planner does not have to consult the higher level planners to resolve the problem. However, obstacle D is directly in the path. In this case, the local planner signals an obstructed local path, and this, in turn, propagates all the way back to the
global planner. The global planner then generates a new global path which avoids all obstacles, and thus the robot reaches its destination. The fact that the robot successfully reaches the destination regardless of these obstacles and difficulties verifies the functioning of the whole system. The time to reach the destination took 2 minutes. The robot calculated for 10 seconds before making the first motion. Thus it is reasonable to state that this system functions quickly enough to be useful in a realistic, real-world environment.

6.3. Analysis of Overhead and Throughput of Blackboard

We developed a series of timing experiments to measure the throughput of various connections in the blackboard. Since the communications links interconnect different processors and operating systems, we expect that the throughput of each link will vary slightly. Since the clocks of each processor are not synchronized, measuring the time of arrival of an object versus the time it was distributed on another processor does not give an accurate measurement of the travel time. Instead, we measured the round trip time of two objects. To measure this, we created a simple loopback program that registers an interest in a SEND_OBJECT, and when it receives one of these it immediately distributes a RETURN_OBJECT. By measuring the time difference between sending a SEND_OBJECT and receiving a RETURN_OBJECT and dividing by two, we get an average time for a single object transfer.

For all of the timing measurements, we used the timing function gettimeofday() on the Unix machine tavi. This function gives a theoretical precision of 1 microsecond. However, the true precision is machine dependent; in the case of tavi, we measured the maximum accuracy to be 20 milliseconds. To improve the accuracy of our measurements, we measured the time for sending an object back and forth 50 times. This
gave us an effective accuracy of 400 microseconds. The objects being transferred each contained 408 bytes of information. We ran each experiment five times and then averaged the results. The following table shows the results for each type of connection and processor:

<table>
<thead>
<tr>
<th>Connection</th>
<th>Transfer Time (ms)</th>
<th>Throughput (bps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tavi &lt;-&gt; tavi</td>
<td>14.84</td>
<td>220,000</td>
</tr>
<tr>
<td>tavi &lt;-&gt; qed</td>
<td>102.00</td>
<td>32,000</td>
</tr>
<tr>
<td>tavi &lt;-&gt; Lynx</td>
<td>21.96</td>
<td>148,600</td>
</tr>
<tr>
<td>tavi &lt;-&gt; Explorer</td>
<td>3234.00</td>
<td>1,009</td>
</tr>
</tbody>
</table>

Table 1. Transfer Time Between Processors

The first experiment we ran measured the time to transfer an object between two Knowledge Sources on the same processor. Since this involves only local communications, we expected this to be relatively fast, which it was. Next we transferred objects between Knowledge Sources on two Sun computers. Each of these processors included its own daemon; thus the path of the object includes transfer to the local daemon, forwarding across the net to the remote daemon, and then locally retransmitted to the other Knowledge Source. This involves more processing steps as well as the network delay and thus runs significantly slower than transfers on the same machine. In contrast, the LYNX processor uses a slave configuration for its KS's; this involves only one direct transfer for objects and thus it is almost five times faster than the daemon to daemon. The improved data rate with the LYNX can also be attributed in part to the real-time kernel. Specifically, the very fast response time of the kernel helps it quickly service events.
The remaining connection that we timed was between the Sun and the Explorer; this connection is significantly different from the others because of the conversion from Unix bytes to LISP structures. The measured times are an order of magnitude slower than on the UNIX transfers; we determined that the majority of the time was spent converting the UNIX bytes to LISP. The turnaround time, the re-conversion (LISP to UNIX) and the communications costs were all comparable to the Sun times. We believe that the extreme delay associated with the conversion is due to inefficient use of the conversion routines built-in to the Explorer. By analyzing the built-in routines of the Explorer more closely, we expect to be able to create a new conversion routine that will reduce the delay to be comparable with the UNIX processors.

The final performance measurement we made on the blackboard was to determine the effect of fan-in and fan-out on the object distribution. Fan-out occurs when several Knowledge Sources register interest in an object and a separate Knowledge Source subsequently distributes this object. In this case, the single object fans out into multiple copies; one for each Knowledge Source. At some point, a separate copy has to be made of the object for each Knowledge Source. Thus we would expect the total time to distribute the object to be approximately equal to the transfer time of an object to one Knowledge Source times the number of Knowledge Sources requesting it. Fan-in occurs when multiple Knowledge Sources distribute a particular object type simultaneously and these objects are requested by a single Knowledge Source. It is difficult to measure either fan-in or fan-out separately for the same reason that it is difficult to measure a single object transfer. Instead we devised an experiment in which we measured both characteristics simultaneously. In this setup, we ran five of the loopback programs previously mentioned concurrently. Then the debugging program distributed the SEND_OBJECT. The debugger recorded the time of arrival of each of the five REPLY_OBJECTs. This was repeated for five repetitions to build up a more accurate
timing. And the whole test was run 5 times and then averaged. The results looked like the following:

<table>
<thead>
<tr>
<th>Process #</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>108</td>
</tr>
<tr>
<td>2</td>
<td>110</td>
</tr>
<tr>
<td>3</td>
<td>112</td>
</tr>
<tr>
<td>4</td>
<td>114</td>
</tr>
<tr>
<td>5</td>
<td>116</td>
</tr>
</tbody>
</table>

*Table 2. Measured Fan-out for 5 Processes*

The fan-out time can be calculated by taking the time of receiving the first `REPLY_OBJECT` and subtracting the transfer time of a single object. We calculated that to fan-out to five Knowledge Sources it took approximately 65 ms; this translates into 13 ms to transfer the object to one of the five Knowledge Sources. This is slightly less than the transfer time of a single object. This slight speedup is due to the fact that object is locally transferred to the daemon only once, and then transferred to the other Knowledge Sources from there. We would expect that since each Knowledge Source received its `SEND_OBJECT` in 13 ms increments, the time between receiving the objects should be similarly staggered. The data in the chart has a smaller increment because the 20msec resolution of the clock made it difficult to accurately measure the 13msec increments.

Depending on which machines the objects are being distributed, the throughput will vary slightly. From these experiments, we can develop a maximum throughput, a minimum throughput and an average. The maximum throughput occurs between objects on the same processor; this is simply 220Kbps. The slowest connection is between the
Explorer and the Lynx (this passes through the Sun); this gives 1Kbps. If we disregard the Explorer connection, the worst throughput is 26Kbps. And the average throughput of the system can be expressed as approximately 111Kbps.

6.4. Response Time of Planners

The next set of timing measurements we made were for each of the planners in the hierarchy. For each of these levels, the debugger started a timer as it distributed the input objects. When it received the results, it measured the time difference. In this case, we timed a single such loop; however, since the times involved in each of these planners are on the order of seconds, this accuracy was adequate. For the global planner, we used the example environment presented above and generated a GLOBAL_GOAL from one corner to the other. This particular path involved twenty two waypoints. The time it took from the distribution of the GLOBAL_GOAL on the Sun to the receipt of a GLOBAL_PATH was approximately 12 seconds. By subtracting the time associated with the object distribution, we see that the actual path generation takes about 6 seconds. For the constraint analyzer, we generated the same GLOBAL_PATH as the global planner created and measured the delay until we received a LOCAL_PATH. This was on the order of 4 seconds. In this case the delays of the blackboard were insignificant in comparison to the computation time of the analyzer. The time between distributing a LOCAL_PATH and receiving a LOCAL_PATH_COMPLETE is about two minutes since the robot actually executes the path. This particular time will vary greatly depending on the maximum speed of the robot and numerous external influences; the only thing we should note here is the order of magnitude of the time. Since we can say that almost any path will take less than 10 minutes to navigate (within a single room), if the other planners run calculations for longer than 10 minutes, the robot will have to
delay its motions. Based upon these timing results, this seems very unlikely. However, we should notice that when the global planner is introduced to a new environment, it takes the program more than 30 minutes to generate the Voronoi diagrams necessary to navigate before any path can be planned.

<table>
<thead>
<tr>
<th>Planner</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>global (Meng's)</td>
<td>6</td>
</tr>
<tr>
<td>analyzer</td>
<td>3.7</td>
</tr>
<tr>
<td>local</td>
<td>110</td>
</tr>
</tbody>
</table>

*Table 3. Path Planner Response Time*

6.5. Interaction of Blackboard and Planners

Having timed the blackboard and each planner separately, we now need to determine how much of the throughput of the blackboard is being used by the planners; this will tell us how much the hierarchy can be expanded before the blackboard becomes a bottleneck in the computations. If we compare the overhead due to the blackboard versus the computation time of each of the path planners, we see that only the UNIX to Explorer connection has a significant impact on the system performance. If we look at the utilization of the connections by each of the planners, we need to estimate how frequently each of the connections is used. The robot will need a global path approximately every 2 minutes. Thus we see that the global planner generates about 500 bytes every 120 seconds, or 33bps. For each un navigable global path, this adds an additional path (33bps). Additionally, we can estimate that the global planner distributes a global map every 5 minutes, which is 80bps. Thus the global planner needs about 150bps. This translates to about 15% of the Explorer to Sun connection. The constraint
analyzer will interact with the local planner a maximum of about 20 times while traversing a single global path (our example setup interacts 2 times). Thus it needs about 700bps for the Sun to Lynx connection. This is 1/2% of the Sun to Lynx connection. Thus the throughput of the system is high enough to allow significant expansion of the hierarchy.
Chapter 7

Conclusions

The goal of this research was to formulate an improved hierarchical path planning architecture and implement it in such a way that it would function on an IAMR system. We demonstrated that creating a hierarchy of planners instead of using a single all-in-one planner can greatly reduce the number of obstacles the planning system must take into account; this, in turn, reduces the computational complexity of the planning while it improves the performance of the robot in real-world environments. We proved that the performance of the classic path planning hierarchy could be significantly improved by adding the constraint analysis level. The constraint analyzers enable the hierarchy to deal with a greater diversity of obstacles and environments without greatly increasing the computational complexity of the system.

By using a distributed blackboard architecture for inter-planner communications we improved the flexibility of the hierarchy and the ease with which it is implemented on the heterogeneous computer environments on IAMRs. Furthermore, by defining a simplified transaction model for this blackboard, we succeeded in removing most of the data consistency problems associated with distributed databases, and at the same time simplified accesses to the blackboard. We further simplified the blackboard by using data-driven scheduling and no history of data. These simplifications did not adversely affect the path planning hierarchy.
Having developed the concepts involved in the hierarchy and the blackboard, we implement a test system using the Rice-obot I. Using this test system we verified that the whole hierarchy successfully navigates a cluttered environment which includes multiple unexpected obstacles, and other obstructions to the path. The measurements we made of the throughput and delays of various parts of the blackboard and planners indicate that the overhead of the blackboard is minor in comparison to the calculation time of the planners in all cases except the Unix to Explorer connection. We believe that the throughput of this connection can be significantly improved with only minor modifications to the LISP implementation of the stub. Further improvement in performance could be gained by using the Explorer II processor which is a significantly faster processor.

There are several areas of further research and improvement in both the hierarchical planning as well as the distributed blackboard. We formulated the architecture of the distributed blackboard system such that it is relatively easy to extend the daemons to use shared memory for local transfers. This extension to the blackboard would make it possible to map hardware buffers (such as video frames) into objects and thus provide extremely high-speed local data transfers. Another important extension to the blackboard would be to implement a daemon on a parallel processor, such as the Sequent. In such a system, the daemon could improve performance by migrating Knowledge Sources among the processors.

Significant improvements on the path planning would result in improving each of the path planners in the hierarchy. The global planner should be expanded to handle a more dynamic global map. When a collision occurs during a rotation on the path, it is possible in many instances to replan that section of the path using rotational path planning techniques to avoid the collision. The technique presented by Jacobs [34] would work well as an additional analyzer. The most dramatic improvements could be
made at the local planner level by adding sensor fusion and map generation capabilities. This could improve the performance of all of the planners by providing accurate map updates and expansion when the robot navigates in unexplored regions. A simpler approach is to use one sensor as the primary sensor, and the others as backups or monitors. Given a local path to traverse, we might begin by using the LIDAR sensors as the primary sensor. If, for some reason, the LIDAR cannot provide a local path (either because of the environment or because it breaks down, etc.) we can revert to the vision. In this way, we can get graceful degradation of performance under failures. By having the blackboard system for communications, we can easily experiment and expand on the system to continuously improve the overall hierarchical system.
References

Blackboards


Distributed O/S and Databases


**General Path Planning**


Rotational Path Planning

32. Brooks, Rodney A., Lozano-Perez, Tomas, "A Subdivision Algorithm in Configuration Space for Findpath with Rotation"


Hierarchial Path Planning


General Mobile Robot Architecture


Distributed & Hierarchial Robot Architectures

57. Butner, Steven Wang, Yulun Mangaser, Amante "Design and Simulation of RIPS: An Advanced Robot Control System", in *Proceedings of the IEEE International Conference on Robotics and Automation*


Appendix A. Implementation Details

9.1. Unix daemon on Suns

The daemon on the Unix machines is implemented using Sockets for both local and global communications. The daemon first reads the hostfile for global connections and attempts to connect to each processor. This establishes the global connections. Then the daemon starts the slave and local listening, and then begins waiting for messages.

```c
main(argc,argv)
char **argv;
{

    /* if there are other daemons out there, connect to them first */

    if (nocall == 0) {
        hostfp = fopen((char *)"hostfile.bbd","r");
        hostndx = 0;
        while (!feof(hostfp)) {
            fscanf(hostfp,"%s",host_table[hostndx++].name);
        }
        fclose(hostfp);
        hostndx--;

        for (i=0;i<hostndx;i++) {
            printf("Attempting connect to %s\n",host_table[i].name);
            serv_chan = open_channel(CH_INET_PORT,CH_STREAM,
                                    host_table[i].name,GLOB_PORT,Handle_Global_Message);
            if (serv_chan != 0) {
                host_table[i].chan = serv_chan;
                printf("successful\n");
            }
            else
                printf("unsuccessful\n");
        }
    }
```
/* start network listening */
fprintf(stderr,"Beginning daemon listening...\n");
con_table[0].chan = serve_channel(CH_INET_PORT,CH_STREAM,
    GLOB_PORT, New_Global_Connect);

/* start slave listening */
con_table[1].chan = serve_channel(CH_INET_PORT,CH_STREAM,
    META_PORT, New_Local_Connect);

/* start local machine listening */
con_table[2].chan = serve_channel(CH_UNIX,CH_STREAM,LOC_PORT,
    New_Local_Connect);
conindex = 3;

/* infinite loop listening for messages*/
while (1) {
    select_dispatch();
}

The select_dispatch routine waits for connections and then when a message is
available, it calls the routine in the read_select_table.

select_dispatch()
{
    read_mask=read_select_mask;
    n=select(32,&read_mask,0,0,tp);
    if(n>0){
        for(i=0, m=1; i<NOFILE; i++, m<<=1)
            if(read_mask & m) {
                ("read_select_table[i].si_handler")
                (i,read_select_table[i].si_params);
            }
        return(n);
    }
}

When a connect message is received, the port address is added to the
read_select_table. If the message is a command from a daemon, then it handles the
various messages as follows:
Handle_Global_Message(fid, chan)  
{
    g_receive(chan,&header,&message,&objhandle,&objsize);
    switch (header) {  
/* a daemon wishes to have objects of objecttype forwarded to it.  
ad it to the links for that object */
    case REGISTER_FORWARD:
        addlink(message,obtype, 
                0,chan,GLOBAL_CALL);
        break;

/* another daemon has sent a message to be forwarded to the local  
Knowledge Sources */
    case FORWARD_OBJEKT:
        obtype = obj_handles[objhandle].obtype;  
        thisLink = call_table[obtype];
        while (thisLink != NULL) {  
/* only forward obj locally. the originating daemon will  
do all of the global calls */
            if (thisLink->type == LOCAL_CALL) {  
                _send(thisLink->chan,  
                       thisLink->event,objhandle,objsize);
            }
            thisLink = thisLink->next;
        }
        break;
    }

9.2. Stub code and Slave code

The stub code handles just the communications with the local daemon. In the Unix  
implementation, the connection is a Socket; the _send routine handles this low-level  
communications using the port established when the code is initiated. The Slave code is
exactly the same as the stub code except that the address used to initiate the daemon connection is different and the processor specified in the connection is the particular host machine.

```c
RegisterInterest(objektype, proctype)
{
    ev_table[evindex].proctype = proctype;
    ev_table[evindex].objtype = objektype;
    message.event = evindex++;
    message.objtype = objektype;
    _send(REGISTER_INTEREST, message, NULL, 0);
}

DistributeObjekt(objhandle, objsize)
{
    _send(DIST_OBJEKT, zernmsg, objhandle, objsize);
}

handle CreateObjekt(objektype, objsize)
{
    objptr = (char *)malloc(objsize);
    obj_handles[hindex].objsize = objsize;
    obj_handles[hindex].objtype = objektype;
    obj_handles[hindex++].objptr = objptr;
    *objptr = (unsigned int)1;

    return((handle)hindex-1);
}

char *AccessObjekt(objhandle)
{
    return((char *)obj_handles[objhandle].objptr);
}

ReleaseObjekt(objhandle)
{
    free(obj_handles[objhandle].objptr);
}
void Objekt_Dispatch()
{
    if (numrec > 0) {
        _receive(&event,&objhandle,&objsize);
    } else {
        /* error!!! */

        /* call the routine with the object handle */
        /* and the object size */
        (*ev_table[event].procptr)(objhandle, objsize);
    }
}

9.3. Object definition

Object definitions in C consist of a unique number for each type of object and an associated structure definition. This information is kept in an include file that the Knowledge Source code uses when manipulating and creating objects. The following are some examples of these definitions.

#define GLOBAL_PATH 0x1010
#define LOCAL_PATH 0x1020
#define MAP_OF_ROOM 0x1030

struct s_global_path {
    unsigned int flags;
    unsigned int numwaypts;
    int wayptx[MAXWAYPTS];
    int waypty[MAXWAYPTS];
};

struct s_global_path {
    unsigned int flags;
    unsigned int numwaypts;
    int wayptx[MAXWAYPTS];
    int waypty[MAXWAYPTS];
};
9.4. LISP code specifics

The current implementation in LISP is based upon the Slave code. Instead of the
C-language structures, the LISP implementation uses LISP structures. These are defined
using the built-in macros available for structure manipulation. Typical structure
definitions look like the following:

```
(defstruct (Str-L-Path (:type :grouped-array 200)
                        (:times 100))
    wayptx waypty)

(defstruct (Str-G-Path (:type :grouped-array 200)
                        (:times 100))
    wayptx waypty)

(defstruct (Str-G-Goal (:type :array 50))
    flags startx starty endx endy orient)
```

For each object type, we also need a conversion routine. These routines convert
an object to a byte buffer and vice versa.

```
(defun buf-to-G-Path (buffer)
  (let ((i 0) retstruc)
    (setq retstruc (make-Str-g-path))
    (setf (str-g-path-wayptx 0 retstruc)
          (char2long buffer 0))
    (setf (str-g-path-waypty 0 retstruc)
          (char2long buffer 1))
    (do ((i 1 (1+ i)))
        ((= i 50))
      (setf (str-g-path-wayptx (* i 2) retstruc)
            (char2long buffer (+ 4 (* i 4)))))

    (do ((i 1 (1+ i)))
        ((= i 50))
      (setf (str-g-path-waypty (* i 2) retstruc)
            (char2long buffer (+ 204 (* i 4)))))
    retstruc)
)
9.5. Interface of Global Planner

The global planner designed by Alex Meng was designed as a stand-alone program; as such it required some modifications to interface to the blackboard. The whole planner is interfaced as a single Knowledge Source. We basically left the graphics and user interface intact; when a global goal is received, the handling routine simply inputs it as if the user had specified it as a start and goal. Then it instructs the planner to plan a path between these points. Finally, it distributes a GLOBAL_PATH object. If an OBSTR_GLOBAL_PATH (obstructed global path) is received, the handler feeds the remaining piece of the path (passed in the object) into the path replanning routine. This generates a new path, which is rebroadcasted as a GLOBAL_PATH. The following code establishes the interests and distributes the global map before any messages are processed.

(defun meng-bb (dummy1 dummy2)
  (let (myhandle myobj))
    ; register interests...
    (RegisterInterest GLOBAL.GOAL 'specify-goal)
    (RegisterInterest OBSTR_GLOBAL_PATH 'handle-obstacles)
    ; distribute the global map for those who may be interested
    (distribute-global-map)
  )
)

9.6. Constraint Analyzer

The constraint analyzer has an interest in global paths and global maps. When it receives a GLOBAL_MAP, it stores it for later use. When it receives a
GLOBAL_PATH, it steps through all of the segments of the path, orients the robot at
each point, and then checks for collisions with surrounding obstacles. If a collision is
found, it creates an OBSTR_GLOBAL_PATH. Otherwise, it creates a LOCAL_PATH
with the same path segments.

```c
main_init()
{
    init_rob_poly(poly.ptr);
    RegisterInterest(GLOBAL_PATH, AnalyzeGlobalPath);
    RegisterInterest(GLOBAL_MAP, NewGlobalMap);
}
```

AnalyzeGlobalPath(objhandle, objsize)
handle objhandle;
int objsize;
{
    obj.ptr = (struct s_global_path *)AccessObjekt(objhandle);

    old[0] = objptr->wayptx[0];
    old[1] = objptr->waypty[0];
    pathseg = 1;

    /* check for a collision in all of the path segments */

    while ((pathseg < objptr->numwaypts) && (coll == 0)) {
        /* position and orientation of point */
        new[0] = objptr->wayptx[pathseg];
        new[1] = objptr->waypty[pathseg];
        dist[0] = (new[0]-old[0])/NUMDIV;
        dist[1] = (new[1]-old[1])/NUMDIV;
        angle = (float)atan((double)(new[0]-old[0])/
                           (double)(new[1]-old[1]));

        if ((new[1]-old[1])<0)
            angle = angle+180;

        /* check for collisions */

        for (i=0; i<NUMDIV; i++) {
            pt[0] = old[0]+(i*dist[0]);
            pt[1] = old[1]+(i*dist[1]);
            coll = check_for_collision(pt,angle,poly.ptr,map.ptr);
```
old[0] = new[0];
old[1] = new[1];
pathseg++;
}
if (coll != 0) {
    printf("A collision was found\n");
    /handle = CreateObjekt(GB_GLOB_10_PATH,
                          sizeof(struct s_obstr_global_path));
    gpath = (struct s_obstr_global_path *)AccessObjekt(handle);
    fill_OGP(gpath, objptr->wayptx[pathseg], objptr->waypty[pathseg]);
    DistributeObjekt(handle);
} else {
    printf("No collisions found so generating local path\n");
    /handle = CreateObjekt(LCL_PATH, sizeof(struct s_local_path));
    lpath = (struct s_local_path *)AccessObjekt(handle);
    for (i=0; i<objptr->numwaypts; i++) {
        lpath->wayptx[i] = objptr->wayptx[i];
        lpath->waypty[i] = objptr->waypty[i];
    }
    lpath->numwaypts = objptr->numwaypts;
    DistributeObjekt(handle, sizeof(struct s_local_path));
}