MIT Libraries
Document Services

Room 14-0551
77 Massachusetts Avenue
Cambridge, MA 02139
Ph: 617.253.5668 Fax: 617.253.1690
Email: docs@mit.edu
http://libraries.mit.edu/docs

DISCLAIMER OF QUALITY

Due to the condition of the original material, there are unavoidable flaws in this reproduction. We have made every effort possible to provide you with the best copy available. If you are dissatisfied with this product and find it unusable, please contact Document Services as soon as possible.

Thank you.

Some pages in the original document contain color pictures or graphics that will not scan or reproduce well.

Stupid Robot Tricks: A Behavior-Based Distributed Algorithm Library for Programming Swarms of Robots

by

James D. McLurkin

M.S., Electrical Engineering
University of California, Berkeley, 1999

S.B., Electrical Engineering
Massachusetts Institute of Technology, 1995

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 2004 [June 2004]

© 2004 James D. McLurkin, All rights Reserved

The author hereby grants to MIT permission to reproduce
and to distribute publicly paper and electronic
copies of this thesis document in whole or in part.

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

Certified
by.....
Leslie P. Kaelbling
Research Director, Computer Science and Artificial Intelligence Laboratory
Department of Electrical Engineering and Computer Science
Thesis Supervisor

Certified
by.....
Dr. David Barrett
Director of Research, iRobot Corporation
Thesis Supervisor

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

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 26 2004
LIBRARIES

ARCHIVES

Stupid Robot Tricks: A Behavior-Based Distributed Algorithm Library for Programming Swarms of Robots

by

James D. McLurkin

Submitted to the Department of Electrical Engineering and Computer Science on May 7th, 2004, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Abstract

As robots become ubiquitous, multiple robots dedicated to a single task will become commonplace. Groups of robots can solve problems in fundamentally different ways than individuals while achieving higher levels of performance, but present unique challenges for programming and coordination. This work presents a set of communication techniques and a library of behaviors useful for programming large groups, or swarms, of robots to work together.

The gradient-flood communications algorithms presented are resilient to the constantly changing network topology of the Swarm. They provide real-time information that is used to communicate data and to guide robots around the physical environment. Special attention is paid to ensure orderly removal of messages.

Decomposing swarm actions into individual behaviors is a daunting task. Complex and subtle local interactions among individuals produce global behaviors, sometimes unexpectedly so. The behavior library presented provides group behavior "building blocks" that interact in predictable manner and can be combined to build complex applications. The underlying distributed algorithms are scaleable, robust, and self-stabilizing.

The library of behaviors is designed with an eye towards practical applications, such as exploration, searching, and coordinated motion. All algorithms have been developed and tested on a swarm of 100 physical robots. Data is presented on algorithm correctness and efficiency.

Thesis Supervisor: Leslie P. Kaelbling
Professor of Computer Science and Engineering, MIT
Research Director, Computer Science and Artificial Intelligence Lab

Thesis Supervisor: Dr. David Barrett
Director of Research, iRobot Corporation

Acknowledgments

"It takes a village to write a thesis"
Ancient African-American Proverb[1]

(In order of appearance)

iRobot has been a great place to work and grow over the past five years. Thank you all for the help and support and patience. Dr. Douglas Gage from DARPA, who has been more than just a program manager. He also shares the dream of "Zillions and zillions of robots". Mike Ciholas was instrumental in getting the first 12-robot swarm up and running. David Sotkowitz, my "spiritual programming guide" improved my coding style tremendously. Jim Frankel, the long-haired engineer responsible for most of the SwarmBot design, and I are alike in so many ways, it always made coming to work fun. He has been a great ally and friend over the years. Ed Williams solved all the hard problems on the robot design. Steve Lacker (the slacker) started us towards production with an inspired mechanical design. Chi Won pulled more all-nighters than anyone ever should to put the robot into production. The Swarm Build Day Crew built 100 robots in 6 hours! Jennifer Smith has been the "other half" of the iRobot Swarm project for three years now, and was responsible for much of the infrastructure the algorithms are built upon. The UROP Heros, Shuang You, and Yuran Yu, who replied to my 11th hour requests for help and saved the day.

This work was supported by DARPA IPTO under contracts SPAWAR N66001-99-C-8513 and SMDC DASG60-02-C-0028.

And finally, Dara Bourne, who has provided support in quantities sufficient to brace the Hoover Dam. This work would have not had been possible without her. Thank you Dara.

[1] Well, maybe not that ancient...

This document is possible because I have had the fortune to work with a long string of wonderful advisors. I dedicate this thesis to them

Mr. Eugene Warasilla 1988-1990
"Ya know, that reminds me of a story..."

Thank you for starting me on this journey.

Dr. Anita Flynn 1991-1995
"Make it happen."

Thank you for being able to simultaneously hold the carrot, wield the prod, and dispense good advice.

Prof. Rodney Brooks 1995-1997
"Fast Cheap and Waaay Out of Control"

Thank you for showing me how crazy people can change the world.

Prof. Kristofer Pister 1997-1999
"Will you quit wasting your time and make a difference out there?!"

Thank you for demonstrating how exceptionally talented engineers solve problems.

Dr. Polly Pook 1999-2002
"You can call me many things, but don't call me Ma'am"

Thank you for believing that I could bring the Swarm to reality.

Prof. Leslie Kaelbling 2001-Present
"That's nice. Where's the data?"

Thank you for showing me how to act like a scientist.

Contents

Chapter 1. Introduction **15**
1.1 Ants, Bees, and other Related Work 16
1.1.1 Related Work 17
1.2 The SwarmBot 19
1.2.1 Sensors 19
1.2.2 ISIS Communication System 20
1.3 SwarmOS 21
1.4 Hands-Off Operation: HIVE™ and the Robot Ecology™ 21
1.5 Assumptions, Design Goals, and Conventions 22
1.5.1 Assumptions 22
1.5.2 Design Goals 23
1.5.3 Conventions 25
Chapter 2. Neighbors and Communications **26**
2.1 The Swarm Neighbor System 26
2.1.1 Neighbor Packet Types 26
2.1.2 Periodic Neighbor Transmit Cycle 27
2.1.3 NeighborOps 29
Chapter 3. Gradient Message Propagation **30**
3.1 Gradient Propagation 32
3.1.1 Normal Gradients 33
3.1.2 Gradients with Lateral Inhibition 36
3.2 Gradient Clean-up 38
3.2.1 Message Clean-up 40
3.2.2 Time-stamp Clean-up 42
3.2.3 Combination Clean-up 45
3.3 Summary 45
Chapter 4. The Swarm Behavior Library **46**
4.1 Behavior Operations 47
4.2 Types of Behaviors 48
4.2.1 Functional Behavior Groupings 48
4.2.2 Metrics 50
4.2.3 Experimental Setup 50
4.3 Primitive Behaviors 51
4.3.1 moveArc 51
4.3.2 moveStop and moveForward 51
4.3.3 moveByRemoteControl 51
4.3.4 bumpMove 52
4.4 Pair Behaviors 52

4.4.1 orientToRobot 52
4.4.2 matchHeadingToRobot 53
4.4.3 followRobot 55
4.4.4 avoidRobot 57
4.4.5 orientForOrbit 58
4.4.6 orbitRobot 60
4.5 Group Behaviors 62
4.5.1 avoidManyRobots 62
4.5.2 disperseFromSource 64
4.5.3 disperseFromLeaves 66
4.5.4 diperseUniformly 69
4.5.5 followTheLeader(-data) 74
4.5.6 orbitGroup 77
4.5.7 navigateGradient 80
4.5.8 clusterOnSource 83
4.5.9 clusterIntoGroups 86
4.5.10 detectEdges(-polish, -data) 89
4.6 Summary 91
Chapter 5. Applications and Demonstrations 93
5.1 Surround Object 93
5.2 The MegaDemo 94
5.3 Lemmings 95
5.4 The Swarm Choir 97
5.5 Directed Dispersion 98
5.6 Summary 102
Chapter 6. Conclusions and Future Work 103
6.1 Limitations 103
6.2 Future Work 104
6.3 Final Remarks 104
Appendices 106
A1. neighborOps Examples 106
A2. Experimental Data – Robot Path Traces 109
References 123

Figures

Figure 1: The iRobot Swarm has over 100 robots 15
Figure 2: Insect communities are superb examples of distributed autonomous systems. 16
Figure 3: This diagram shows the interactions that honey bees use to determine foraging recruitment. 17
Figure 4: The iRobot SwarmBot™ is been designed for embodied distributed algorithm development 19
Figure 5: The iRobot ISIS™ system allows each robot to communicate with its neighbors and determine their range, bearing, and orientation 20
Figure 6: The Robot Ecology™ provides resources for autonomous charging and navigation. 21
Figure 7: Diagram conventions used in the behavior descriptions. 25
Figure 8: A communications gradient is formed as messages are relayed from robot to robot. 32
Figure 9: The normal gradient compare function will tessellate the swarm into Voronoi cells 34
Figure 10: Gradient messages are buffered for a short time 35
Figure 11: Gradient latency data from propagation trials. 36
Figure 12: Sources of gradients with lateral inhibition override other sources 37
Figure 13: Gradients with lateral inhibition can be used to elect a leader 38
Figure 13: Gradients with lateral inhibition can be used to elect a leader 38
Figure 14: Gradient message clean up is as important as propagation. 39
Figure 15: The ideal gradient clean-up function. 39
Figure 16: Clean up messages can remove a gradient in the theoretical minimum time bound 40
Figure 18: Time Stamp Cleanup eliminates back-propagation of gradients. 43
Figure 19: Time-stamp cleanup data. 45
Figure 20: The static function call tree of the Swarm Behavior Library 48
Figure 21: State-of-the-art data collection hardware 50
Figure 22: The `orientToRobot` behavior – diagram. 52
Figure 23: The `orientToRobot` behavior – experimental resuls. 53
Figure 24: The `matchHeadingToRobot` behavior when used with the ISIS beacon forms a compass. 54
Figure 25: the `matchHeadingToRobot` behavior – video images 54
Figure 26: The `matchHeadingToRobot` behavior – experimental results 55
Figure 27: The `followRobot` behavior – experimental results. 56
Figure 28: The `avoidRobot` behavior – experimental results 57
Figure 29: The `orientForOrbit` behavior - diagram 59
Figure 30: The `orientForOrbit` behavior – experimental results 60
Figure 31: The `orbitRobot` behavior - diagram 60
Figure 32: The `orbitRobot` behavior – video captures 61

Figure 33: The orbitRobot behavior – experimental robot paths 61
Figure 34: The orbitRobot behavior – experimental results 62
Figure 35: The avoidManyRobots behavior - diagram 62
Figure 36: The avoidManyRobots behavior – experimental results 63
Figure 37: The disperseFromSource behavior – diagram 64
Figure 38: The disperseFromSource behavior – experimental results 65
Figure 39: The disperseFromSource behavior – video images 65
Figure 40: The disperseFromLeaves behavior – diagram. 67
Figure 41: The disperseFromLeaves behavior – experimental results 68
Figure 42: The diperseUniformly behavior - velocity equations 70
Figure 43: The diperseUniformly behavior – video images 71
Figure 44: The diperseUniformly behavior uses an approximation to compute Voronoi neighbors 72
Figure 45: The diperseUniformly behavior - experimental results 73
Figure 46: The diperseUniformly behavior – pictures 73
Figure 47: The followTheLeader behavior - diagram 74
Figure 48: The followTheLeader behavior – video images 76
Figure 49: The orbitGroup behavior produces the 78
Figure 50: The orbitGroup behavior – experimental results 80
Figure 51: The navigateGradient behavior - diagram 81
Figure 52: The navigateGradient behavior – experimental results 82
Figure 53: The navigateGradient behavior – video images 82
Figure 54: The navigateGradient behavior – robot traces 83
Figure 55: The clusterOnSource behavior – video images 83
Figure 56: The clusterOnSource behavior – diagram 84
Figure 57: the clusterOnSource behavior – robot path traces 85
Figure 58: The clusterOnSource behavior – path efficiency results 85
Figure 59: The clusterOnSource behavior – packing efficiency results 86
Figure 60: The clusterIntoGroups behavior – video images 88
Figure 61: The clusterIntoGroups behavior – experimental results 89
Figure 62: The detectEdges behavior – diagram. 90
Figure 63: The Surround Object Demo – video images 93
Figure 64: The MegaDemo Application 94
Figure 65: The Lemmings Language lets younger robotisists program the Swarm. 95
Figure 66: The inspiration for the Lemmings demo. 96
Figure 67: A solution program for a lemmings maze 96
Figure 68: The Swarm Choir 97
Figure 69: Directed Dispersion frontier robots guide the swarm into unexplored areas.. 99
Figure 70: Directed Dispersion pictures 101
Figure 71: Dispersion efficiencies of five algorithms. 101

Tables

Table 1: Gradient Message Struct Public Members (transmitted to neighboring robots)31
Table 2: Gradient Message Struct Private Members (not transmitted to neighboring robots) 31
Table 3: The behaviorOutput struct members. 47
Table 4: Follow the leader input parameters. 74
Table 5: Follow the leader neighbor data byte variables. 74
Table 6: Summary of behavior performance. 91

Chapter 1. Introduction

The most desirable applications for robots are jobs that are dangerous, dirty, or dull. Many of these jobs lend themselves to being performed by groups of robots working together rather than by single robots working alone. Some tasks can achieve efficiency gains as a direct function of the number of robots applied. Other applications can benefit even more, as radically different techniques can be employed to solve a problem with ten thousand robots than with ten. As robots become more commonplace, the shift to multiple-robot systems will become the rule, rather than the exception. Engineering large multi-robot systems is unachievable without understanding the complex relationship between individual actions and group behaviors.

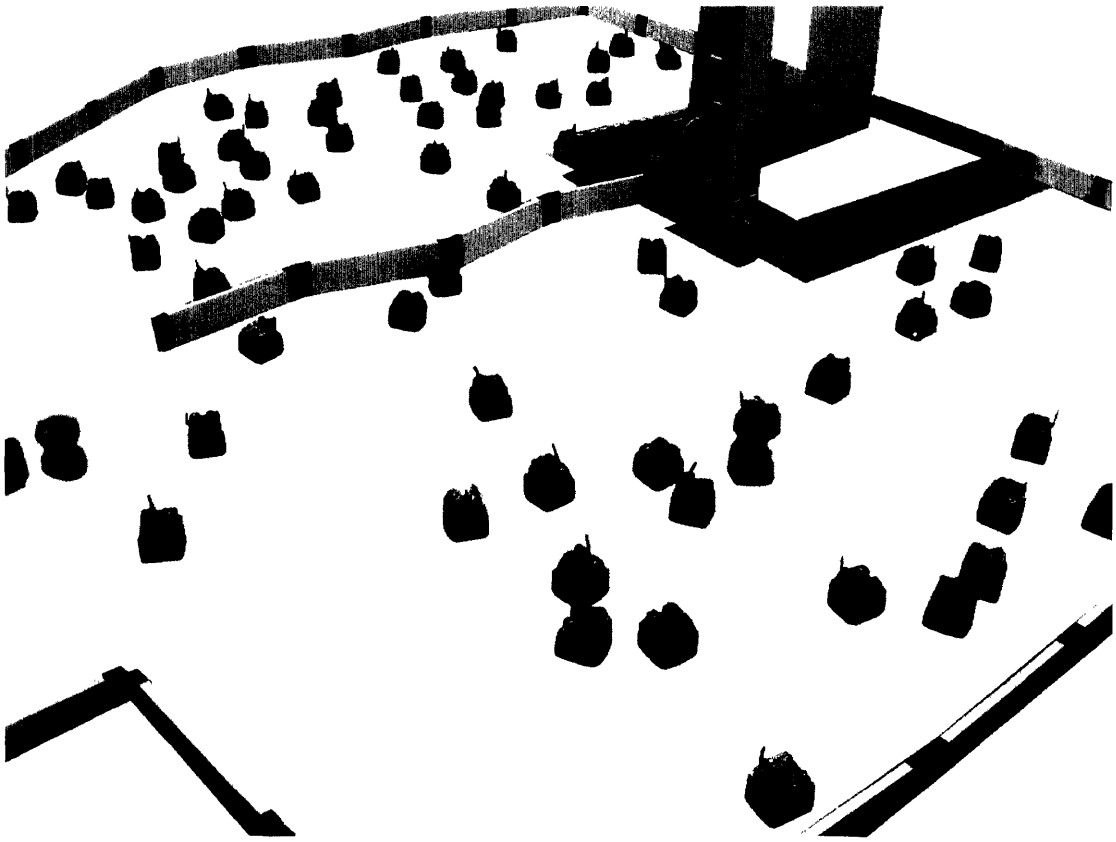


Figure 1: The iRobot Swarm is composed of over 100 individual robots that work together to accomplish group goals.

The goal of this work is to develop distributed algorithms for robotic swarms composed of hundreds of individual robots. Ultimately, we want to be able to write software for large numbers of robots at the group level. The software development system would then compile these group programs into behaviors for individual robots to run. Unfortunately, this top-down approach is a very challenging problem, so we are approach swarm software design from a bottom-up perspective in this work. By designing group-behavior building blocks that can be recombined and reused in many different programs, we will have a swarm programming toolkit that can be used to construct complex global behaviors.

1.1 Ants, Bees, and other Related Work

Biological inspiration is a common theme in robotics. The eusocial insect communities of ants, bees, and termites provide a nearly inexhaustible supply of working algorithms and proven system designs that can be applied to robotic swarms. This work has been heavily influenced by natural systems, from sensor design to software development and everything in between.

The hypothesis is that robots designed with sensors, actuators, and communications that are similar to those of their natural counterparts will also have similar constraints on how they perceive and interact with the world around them. If the problems we want our robots to solve are similar to those solved by insects, and they often are, then algorithms developed for insect survival can be used for inspirations, design guides, and ultimately even for direct comparisons in performance.

These natural systems produce amazingly complex group behaviors from the interactions of thousands, and in some cases millions, of individuals. Figure 3 shows a model of honeybee foraging recruitment. The arrows represent information pathways between forager bees who work outside the hive, food-storer (worker) bees that work inside the hive, and other bees in the hive who observe the recruitment dance of the returning foragers. This information pathway model is very much like a software

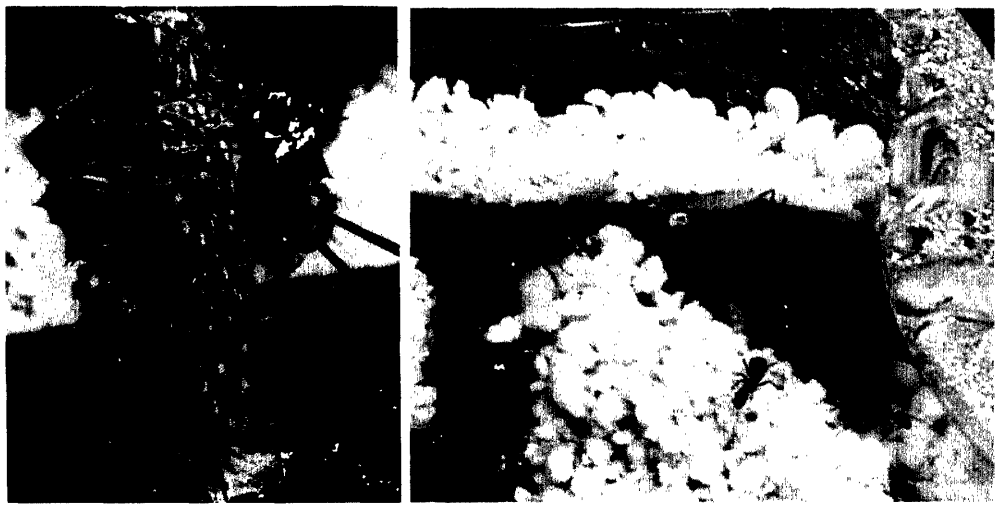


Figure 2: Insect communities are superb examples of distributed autonomous systems. The picture to the left is two leaf cutter ant major workers (*Atta sexdens*) cooperating to cut through a twig. On the right is the author's colony of carpenter ants (*Camponotus pennsylvanicus*).

flowchart. With a capable swarm of robots, it would be possible to model these interactions and simulate honeybee foraging behavior. This could provide a starting point for software design on a team of search-and-rescue robots.

Perhaps even more importantly, the differences between natural and artificial systems can be used to learn more about both. Man-made systems are easy to modify and can collect detailed information about internal state, but they need to be built and programmed before they can be used. Functional biological systems already exist, but they are difficult to modify and it is almost impossible to collect data about their internal state. The duality of these two systems can lead to interesting collaborative experiments. For example, a robotic system based on Figure 3 might work with the removal of an information pathway, or require the addition of a new pathway. This insight could be used by biologists to reinvestigate their models and plan future experiments. This cycle of information exchange could lead to breakthroughs in both fields.

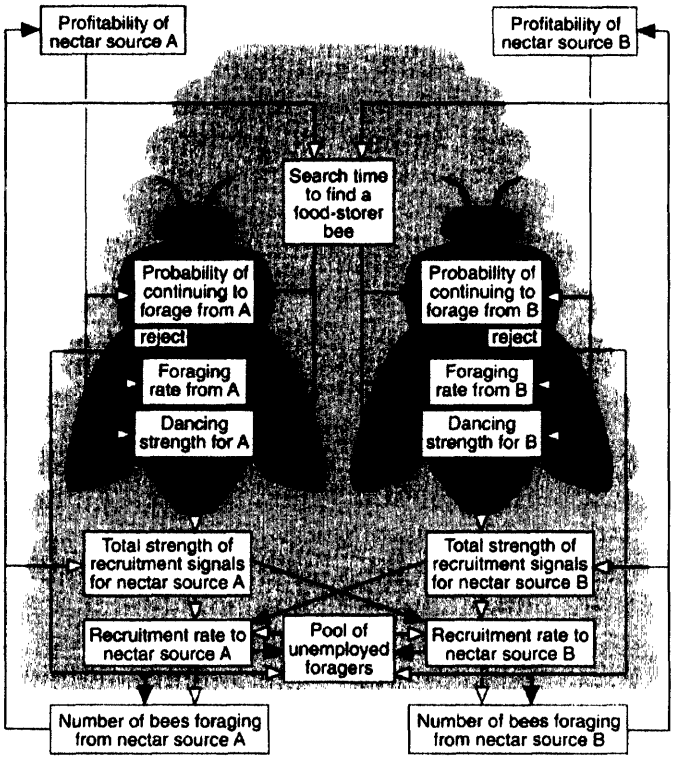


Figure 3: This diagram shows the interactions that honey bees use to determine foraging recruitment [49]. With the addition of a few semicolons, this could be robot software. This represents an exciting new area of research, where we can test biological behavioral algorithms on physical robotic systems

1.1.1 Related Work

Fundamentals

There is a growing literature on distributed algorithms for groups of robots. Much of the work starts with a behavior-based system [Brooks 1985, 1989], which might include various high-level arbiters [Balch/Arkin 1999]. Our interest is focused on large-scale communities with more than ten agents, such as those in [Mataricc 1994] and [McLurkin 1995]. Some form of interrobot communication is required for distributed algorithms. Infrared systems such as those in [McLurkin 1995] and in [Hu/Kelly/Keating/Vinagre 1998] also provide special location information. The radio

system in [Mataricc 1994] provides a global positioning system using stationary beacons as reference points.

An engineering issue that affects algorithm development is the use of unique IDs on the members of the group. The set of algorithms that do not require unique IDs is a proper subset of the total set. Insects do not seem to have global names, but can discriminate between local neighbors. Some researchers [Balch/Arkin 1999], use globally unique IDs, while others argue that local IDs are sufficient.

Algorithm Building Blocks

Much work has been done on motion in formation. Some assume a homogeneous groups of robots, [Balch/Arkin 1999], [Hu/Kelly/Keating/Vinagre 1998], while others assume a leader robot using a more traditional AI path planning algorithm, such as [Desai/Kumar/Ostrowski 1999]. If you have a network of stationary agents, you can "grow" shapes by running programs that make each node change their behaviors [Coore 1999] or even fold origami [Nagpal 2001]. Division of labor is an important part of a multi-agent community. Some [Balch/Arkin 1999] refer to this as "Functional Heterogeneity", emphasizing the point the differences are only in the current behavior of the agents, the hardware is the same. [Mataricc 1998] showed division of labor by robot interactions, then [Schneider-Fontan/ Mataricc 1998] with position information. The biology literature has many examples of division of labor, and computational models of the process have been proposed by [Bonabeau/ Theraulaz/Schatz/Deneubourg 1999] and [Bonabeau/Sobkowski/Theraulaz/Deneubourg 1999]. [Seeley 1995] synthesizes years of work on the communications pathways inside a honeybee colony, and presents computational models for task allocation. Learning is discussed by [Hu/Kelly/Keating/Vinagre 1998] and [Mataricc 1998], and planning is discussed by [Chun/Zheng/Chang 1999], but neither topics are of initial interest to the swarm project. Storing algorithmic state in the physical world as a computational tool has been discussed in [Russell 1995] and [Werger/Mataricc 1996]. The former used a chemical trail, while the latter used the robots themselves as landmarks.

Applications

One of the best tasks for a distributed group of robots is search and mapping. Distributed maps have been made by [Burgard/ Fox/Moors/Simmons/Thrun 2000] and [Yamauchi 1998], both with different strategies for expanding the frontier of exploration. The former explores in the areas of the best rewards, while the later utilizes a "frontier-based" approach to seek out the boundaries between the explored and the unexplored. A comparison between random search and coordinated search can be found in [Gage 1993].

Coordinated manipulation of the environment is another useful task. Pushing a box across a room was explored by [Mataricc 1995] with legged robots and [Parker 1999] with wheeled robots and her Alliance Architecture for interrobot coordination. [Kube/Bonabeau 1999] also demonstrate an algorithm using wheeled robots, but offer a offers a survey of the biological literature on cooperative transport as motivation for the demonstration. Ant colony optimization has proven to be a useful algorithmic technique, and [Botee/Bonabeau 1998] have explored improving it by using genetic algorithms for parameter determination.

Centralized Programming

The ultimate goal it to be able to program distributed systems of many individuals at the group level. [Brooks 1989] speaks of emergent behaviors from a group of behaviors running on a single robot. [Mataricc 1994] extends this to multi robot

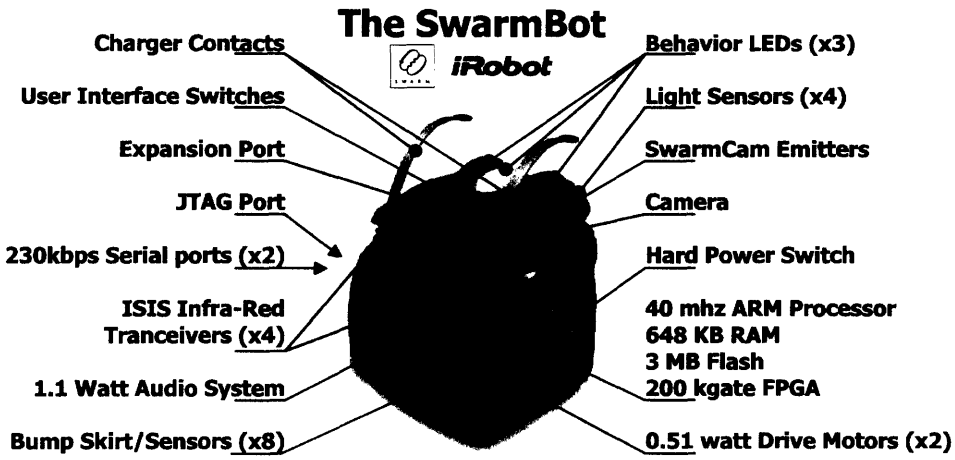


Figure 4:The iRobot SwarmBot™ is been designed for embodied distributed algorithm development. Each robot contains a suite of sensors, inter-robot communication and localization, and a 32-bit microprocessor.

groups, with behaviors interacting across robot boundaries and more complex group behaviors being constructed from simpler behavior primitives. [Coore 1999], [Abelson et al. 1999], and [Nagpal 2001] demonstrate impressive programming systems that take global input, operate in a distributed fashion, and produce global output. The former will grow an arbitrary two dimensional shape, while the later generates origami! A topic with little work, save [Gage 1995], is the management and development infrastructure for a distributed system of physical agents.

1.2 The SwarmBot

The iRobot SwarmBot™[2] shown in Figure 4 has been designed from the ground up for development of distributed algorithms in large swarms. It has a 32-bit RISC ARM Thumb microprocessor, a suite of sensors, good mobility[3], and inter-robot communication and localization. Each robot is 5" on a side, and the total swarm has over 100 units.

1.2.1 Sensors

The SwarmBot has a large sensory suite, including bump sensors, light sensors, a camera, drive-wheel encoders, and the ISIS™ infrared communication and location system. An optional sensory board has been tested that provides a linear CCD and a magnetic "food" sensor. All of the algorithms in this work use only the wheel encoders, the bump sensor, and the ISIS communication system.

[2] iRobot, ISIS, SwarmBot, SwarmOS, HIVE, and "Robot Ecology" are trademarks of iRobot, inc.
[3] In laboratory environments: indoors, on low-pile carpet.

Wheel Encoders

The SwarmBot has four wheels and uses skid-steering to turn. Slippage while turning and the small size of the wheels introduces considerable odometry errors, making dead-reckoning useful for only short distances.

Bump Sensor

The exterior shell of the SwarmBot is an articulated bump sensor. It can detect deflections in the horizontal plane as well as rotations around the center of the robot. This is the robot's primary sensor for obstacle avoidance. Its design guarantees that robots cannot become entangled in each other, but its squareish shape, while stylish, can make it more difficult to negotiate tight spaces.

1.2.2 ISIS Communication System

The iRobot ISIS™ communication and robot location system allows each robot to communicate with its neighbors and determine their range, bearing, and orientation. Figure 5 shows the data made available by the system, and the definitions of range, bearing and orientation. Each robot has an array of twelve IR emitters, grouped into four quadrants. Data can be transmitted from these quadrants independently or in any group. There are four receivers on each robot, which allow it to determine neighbor positions by comparing the signal strengths of one message that is received on two different receivers. Range and bearing are accurate to within 2 cm and 2° at 50 cm of separation. Orientation of the transmitting robot can be computed directly with an accuracy of 45° by observing which emitters the signals originate from. Orientation of the sender measured from the receiver is the same angle as the bearing of the receiver measured from the sender, which allows orientation to be computed using a reciprocal technique that adds one neighbor cycle round-trip communications delay (250 ms, see sec. 2.1.2) but increases the resolution to 2°.

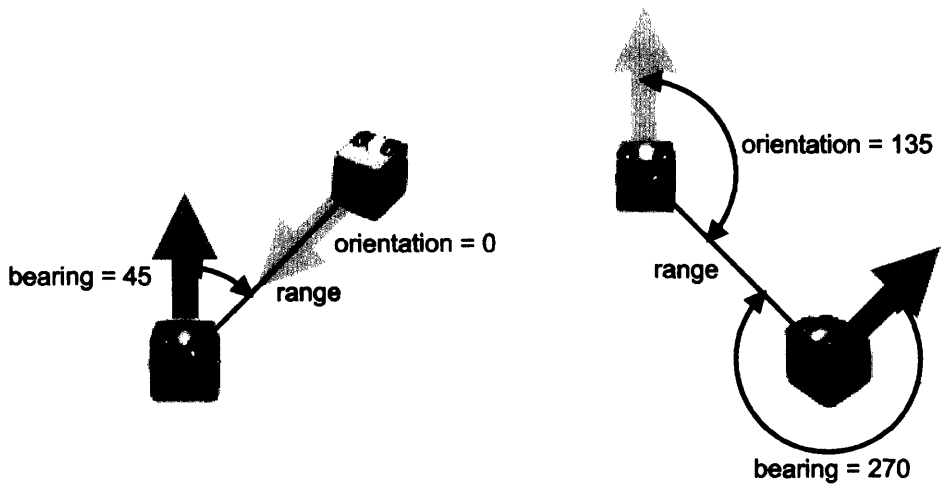


Figure 5: The iRobot ISIS™ system allows each robot to communicate with its neighbors and determine their range, bearing, and orientation. Range is measured from the center of one robot to the center of another. Bearing and orientation are defined relative from one robot to another. In these figures, the bearing and range of the top robots are measured from the bottom robots. We use the term heading to define the orientation of the robot relative to a global external reference frame.

The system has a maximum range of 3 meters, but is typically run at reduced power levels to limit the effective range to about 1 meter. The variable power control allows group experiments to be performed in small laboratory environments while ensuring that any single robot can only communicate with a small number of neighbors.

The ISIS communications system runs at 125 kbps, but packet headers and DC-balanced encoding reduce the throughput to 53.3 kbps with eight byte packets. A FPGA handles all the encoding, transmitting, receiving, and decoding. Each packet has a CRC to ensure data integrity and corrupt packets are detected and discarded by the FPGA hardware. The higher-level communications layer must be able to recover from these losses. Data integrity is quite good over point-to-point communications, with error rates below 0.1%. Multiple transmitters in close proximity create the risk for collisions. The collision problem is discussed in section 2.1.2.

1.3 SwarmOS

The Swarm Operating System (SwarmOS™) provides an API for developers writing applications for the SwarmBot. It was developed at iRobot [1] and controls low-level SwarmBot I/O including: motor control, ISIS drivers, power management/charging, sensor drivers, and remote downloading for wireless software updates. It incorporates the ThreadX real-time kernel from Express Logic [2], which provides a multitasking kernel with an API similar to POSIX. If supports threads, semaphores, mutexes, message queues, and memory allocation. It is designed for embedded applications and has real-time performance and a small memory footprint.

1.4 Hands-Off Operation: HIVE™ and the Robot Ecology™

To work with a large swarm of robots effectively, the user cannot manually program, charge, or even turn on all the robots. Software development, debugging, and analysis must also be performed in a hands-free centralized fashion, without having to physically interact with each robot. The Robot Ecology™ shown in Figure 6 provides resources the robots need to keep themselves running, and the HIVE user interface provides centralized command and control of the swarm. For large swarms, these are

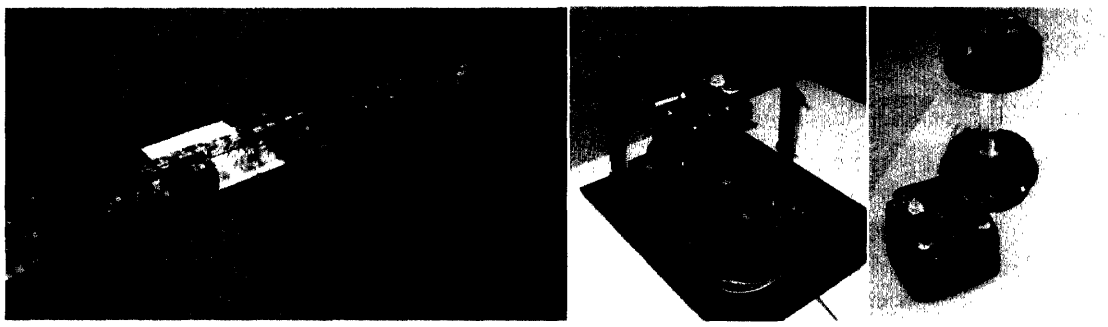


Figure 6: Working with a large swarm of robots requires them to be as self-sufficient as possible. The Robot Ecology™ provides resources for autonomous charging and navigation. **Left:** Chargers allow robots to dock and recharge. **Middle:** Semi-automated testing Allows quick diagnosis of problems. **RIght:** Long-range ISIS beacons aid navigation.

requirements, not luxuries.

1.5 Assumptions, Design Goals, and Conventions

Any scientific work would not be possible without a healthy set of assumptions to reduce the problem to manageable size and a set of design goals to provide direction. Diagrammatic conventions are described at the end of this section.

1.5.1 Assumptions

Local Communications

The assumption that all inter-robot communications are short range is the most important one in this work. It has two important implications:

1. Robots can only communicate with a small subset of the total swarm.
2. There is a relationship between network connectivity and spatial location

The first implication is a powerful tool for ensuring the scalability of the distributed algorithms used by the robots. Physical constraints place a strong upper bound on the number of neighbors an individual robot can have: it is the number of robots that you can pack into communications range. If you assume that robots limit processing to only their local state (see design goal below), this places an upper bound on the memory and processing requirements for each individual. This upper bound is a function of neighbor count, not the total number of robots in the swarm, allowing the swarm to grow and shrink while the demands on individual robots remain constant.

The second implication is that the number of hops a communications packet must take to propagate from one robot to another is related to the physical distance between them. The exact spatial relationship depends on the physical layer of the communication network. The ISIS system uses infra-red light (line-of-sight) and has been designed to provide a uniform, omni-directional transmission pattern. The relationship between network connectivity and spatial location is used throughout this work.

Reliable Lossy Communications

It is assumed that the FPGA firmware and SwarmOS will discard corrupt communications packets, so any message received by the high-level algorithm is a valid message. Additionally, the probability of a successful transmission of a message from one robot to another is assumed to be random and independent of the success of any previous transmission. This implies that there are no systematic errors, and the odds of a single robot not receiving any communications decreases exponentially. However, individual message losses are common and must be tolerated by the software.

Unique ID Numbers

It is possible to divide distributed algorithms into three sets based on the scope of unique identification they require of the agents that run them: global IDs, local IDs, or no IDs. Each set of algorithms is a proper subset of the one preceding it. In order to facilitate development of the largest possible set of algorithms on the swarm, each robot has a 64-bit ID chip, giving each member a globally unique ID.

At the lowest level, individual robots need locally unique IDs to disambiguate communications from nearby neighbors. At the global level, the centralized controller

needs to be able to address each robot individually. Attempts have been made to minimize the scope of unique IDs, but not to eliminate their use. From an engineered-system point of view, the cost and size of ID chips, or a start-up procedure in which all agents select globally unique IDs, is not a prohibitive assumption. This is a potential departure from the natural inspiration, as it is not clear how insects identify their nestmates, but it is probably not with a globally unique identifier. However, insects are able to differentiate amongst neighbors in actions like sharing food and tandem following [10], so locally unique identifiers seem reasonable.

Robust Low-Level Obstacle Avoidance

All the algorithms presented in this work assume that there is some kind of low-level obstacle avoidance behavior that is always successful in guiding the robots away from nearby obstacles. The SwarmBot uses its ISIS system and an array of bump sensors to detect obstacles. The ISIS obstacle detection is not very reliable, but has a range of 10-20 cm. The bump sensors require the robot to collide with an obstacle to sense it, but are very reliable. Once an obstacle has been detected, we assume that the obstacle avoidance behavior will be able to dislodge the robot. In practice, this behavior is almost always successful, with only occasional rescue intervention required to free trapped robots.

1.5.2 Design Goals

Scalability and Robustness

Scalability and robustness often travel hand-in-hand. A robust algorithm will function correctly even if an arbitrary number of robots are removed from the swarm, and a scalable algorithm will function correctly even if an arbitrary number of robots are added to the swarm. In both cases, the swarm must adapt at a global level by responding to its new size. Other changes in the environment, such as erroneous sensory inputs or network failure, must be handled by the swarm as well. This requires that all the distributed algorithms be self-stabilizing, meaning that you can start from any initial state of sensory inputs and robot positions and the system will always converge onto the desired final state. The only limitation we require is that the robots must all be part of the same connected component.

At the local processing level, scalability also requires that algorithms do not scale in running time or in memory space as a function of $\mathbf{n}$, the total number of robots. Most of the algorithms presented scale as a function of the number of neighbors each robot has. The number of neighbors any one robot can communicate with is constrained by the range and bandwidth of the ISIS communications system. Bandwidth is the scarcer resource, limiting the number of neighbors to 20-30, depending on how many messages are sent by the application software. These limits guarantee that neighbor count cannot be a function of total swarm size.

However, making the communications range small increases the number of times a message must be relayed to propagate from one end of the swarm to the other. The swarm can be viewed as a network graph $\mathbf{G}$ with robots as vertices and neighbor communication links as edges. The number of times the message must be relayed (the number of "hops") is equal to $\mathbf{diam(G)}$, the diameter of graph $\mathbf{G}$. Because ISIS is a line-of-sight optical system, $\mathbf{G}$ will correlate strongly with the physical positions of the robots. To compute $\mathbf{diam(G)}$ the exact physical placement of each robot must be

known. In most environments, the distribution of robots can be approximated with a circle, and $\mathbf{diam(G)}$ will grow with order $\mathbf{O(\sqrt{n})}$. Robots arranged in a long, skinny, graph will take longer to propagate information, with the worst case being $\mathbf{O(n)}$ for a line of robots. Lines of robots can be useful for communications relays, but uniformly dispersed robots work well for most other applications.

Homogeneous Hardware and Software

There are many applications of swarms that would require systems of robots with heterogeneous hardware. Some robots could have specialized sensors, some could carry heavy objects, while others could have long-range communications hardware. However, heterogeneous hardware increases design complexity and reduces system robustness.

It is much easier to design and maintain a swarm of homogeneous robots than of heterogeneous ones. The fixed costs of design time, debugging procedures, and spare parts are minimized when amortized across as large a population as possible. Algorithmic robustness is easier to achieve when any robot can perform the role of any other. With heterogeneous hardware, there must be sufficient numbers of each type to ensure that failures can be tolerated. For these reasons, we impose the design constraint of homogeneous hardware and software, but allow robots to changes tasks as needed.

Minimal Local State

Whenever possible, attempts are made to minimize the amount of state each robot needs to maintain, and instead base most local decisions on the current state of sensors and recent communications with neighbors. This is the essence of behavior-based programming, which emphasizes robot control that is tightly coupled to sensory inputs. One of the advantages of this approach is that software must be designed to determine its context from external cues. This allows robots to join or leave the network asynchronously, without having to be told what the rest of the swarm is doing, and with minimal disturbance to the robots around them.

Frequent Communications and Sensing

Frequent communications goes hand-in-hand with maintaining minimal state. In the Swarm, most information about neighbors and the environment has very short time-outs, usually not greater than one second. The rate of communications and sensing must be high enough to keep the robot up-to-date with the current world state. The cost is a large amount of data retransmission, even for unchanging data. The benefit is large design simplification in the rest of the system, as communications, algorithms, behaviors, and neighbor position sensing all become less complex.

Minimal Tuning

Every attempt is made to minimize the number of parameters that require tuning for environmental conditions such as density of robots, number of walls, communications range, etc. Algorithms with sub-optimal performance but fewer "knobs" to turn are preferable to those that can run faster, but require custom fitting for each application. The goal is to make a tool-kit of general-purpose algorithms and behaviors that can be combined and recombined easily to test new ideas.

Modest Local Processing Power

The desire to scale these algorithms to very large swarms of robots implies that they will run on small, cheap microprocessors with limited computational power, at least for the foreseeable future,. The algorithms have been designed to keep processing and memory requirements low. Some of this comes for free by adhering to design principles

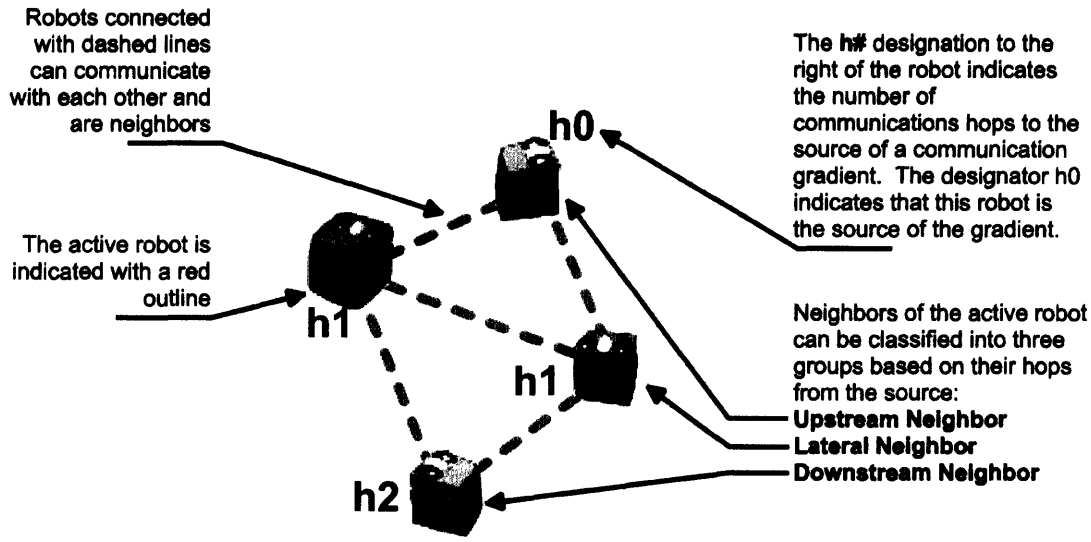


Figure 7: Diagram conventions used in the behavior descriptions. "Upstream" neighbors are one hop closer to the gradient source. "Downstream" neighbors are 1 hop further. It is not possible for any robot to have a neighbor that has more than a one hop difference from itself.

that ensure scalability – memory and processing requirements will only grow as a function of neighbor count, not total number of robots. Other processing efficiencies come from accepting solutions that are simple, but have some inherent inefficiencies, such as the `orbitGroup` behavior from section 0 that guides robots along a suboptimal, but easy to compute, path. The C programming language was used to produce small, efficient machine code. Dynamic memory allocation was outlawed, and the use of floating-point arithmetic was minimized. These procedures allowed the algorithms to run on the prototype SwarmBots which had a 16 mHz 8-bit 6811 microprocessor with 32k of RAM[4].

1.5.3 Conventions

Figure 7 shows the diagrammatic conventions used to describe the algorithms. Graphical icons represent robots. The front of the robots has a slightly contoured appearance, and the three colored lights are on the back. Robots represent vertices of the network graph G, and the communication lines between them are the edges. The hops for the gradient communication algorithms described in Chapter 3 are indicated by a number preceded by the letter "h".

[4] This was before we got spoiled with the snazzy 32-bit systems in the current SwarmBot. The core algorithms remain unchanged, but we surrounded them with MIDI file playback, a slick VT100 terminal interface, and all kinds of other frivolous software. Engineers will be engineers...

Chapter 2.
Neighbors and Communications

The Swarm Neighbor System is responsible for keeping track of neighboring robots and any data they transmit locally to each other. The gradient communication system is built on top of this infrastructure and allows communication messages to travel further than one robot away. However, unlike a standard multi-hop communication system, gradient communications also perform distributed computation as they travel from robot to robot.

2.1 The Swarm Neighbor System

The ISIS infrared communications system is used for all inter-robot communications. The Swarm Neighbor System API creates an easy-to-use abstraction on top of the ISIS drivers, and enforces low-level communications constraints. The application programmer is presented with a shared-memory model of neighbors, their current positions, and their most recent communication messages. This section describes the neighbor system, and some important low-level implementation details that affect algorithm design.

2.1.1 Neighbor Packet Types

There are two types of messages in the neighbor system: neighbor messages and gradient communication messages

Neighbor Messages

Neighbor messages are used to determine range, bearing, and orientation between neighboring robots. The implementation details of the positioning system is not important for understanding the algorithms presented, but some details are worth noting. The resolution of the bearing and orientation is quite good, about 2° at 50 cm of separation. This resolution is useful to avoid discontinuities in the inputs to the many control loops that respond to bearing and orientation changes. The resolution of the range information is about 2 cm at 50 cm of separation. However, the range measurement can be quite noisy, and care must be taken to process the data to avoid chatter in higher-level software.

Each neighbor message contains the sender's robotID, low-level ISIS positioning information, and an arbitrary number of bytes of general purpose data called neighbor variables. Some uses of neighbor variables are to communicate current job, current leader, relevant sensory data, etc. Neighbor variables are global variables that are broadcast each neighbor cycle. In the pseudocode, the syntax is as follows:

```
clusterIntoGroups(beh)⟨groupGradientType⟩
defineNbrVar ⟨grouped⟩
```

This function has as an input one normal variable, **beh**, and one neighbor variable **groupGradientType**. It also "creates" a neighbor variable **grouped** with local scope to this function. It can then read and write from these neighbor variables like any other variable. In addition, it can read the state of any neighbor variable from any current neighbor. The pseudocode to read the state of the **grouped** variable from neighbor **nbr** and determine if it has the value TRUE would be:

```
if(nbr.grouped = TRUE)
```

In actual code, all neighbor variables are passed in as pointers so the behavior functions can be re-entrant, but this level of detail clouds the exposition of the algorithms.

The number of actual ISIS communication packets in each message increases as the number of neighbor variables increases. It is important to minimize the number of packets sent, as sending too many will degrade inter-robot communications. The relationship between the number of packets sent and network degradation is discussed in section 2.1.2 below.

Gradient Communication Messages

Gradient messages carry data and routing information that allows them to be relayed from robot to robot. Each message contains the gradient type, the source robotID, the sender's robotID, the number of times this packet has been relayed (communication "hops"), a time stamp, and three bytes of data. The details of how these packets are relayed, what computation occurs at each step, and how the results are used is an integral part of the distributed algorithms, and is discussed in detail below.

2.1.2 Periodic Neighbor Transmit Cycle

Synchronous System Model

The neighbor system strengthens the design goal of section 1.5.2 from frequent communications to periodic communications. Since all the robots share the same transmission period, every robot will receive messages from each of its neighbors only once per period. This is not limited to neighboring robots, it is not possible for any robot to receive more then one message from any other robot during one period, because that would require the transmitting robot to have a shorter transmission period. This makes the programming model for the swarm appear to be a synchronous distributed system from each robot's point of view. This greatly simplifies algorithm design and validation, because it is possible to place an upper bound on the time at which you should receive a message from a neighboring robot. These time bounds can then be extended to include the entire swarm, permitting stronger conclusions and allowing some classic distributed algorithms to be adapted to robotic applications.

Communications Throughput and Message Collisions

SwarmBots use the ISIS communication system to broadcast their externally visible state omnidirectionally to all nearby robots. The ISIS system supports a Carrier Sense, Multiple Access (CSMA) network, and robots do not transmit while they are

receiving data. Robots maximize their ability to share the communication channel by sending bursts of neighbor messages at periodic intervals.

Although every robot is transmitting messages in periodic bursts and the ISIS system can sense when the channel is in use, there is no centralized controller assigning time slices, so message collisions are still possible. This is very similar to the Aloha protocol [6], which demonstrates a practical channel usage of about 50% of the channel bandwidth. The inter-robot communications bandwidth is the most important design constraint in the system.

As much of the motion of any robot is based on the positions of its neighbors, the periodic retransmission rate needs to be fast enough to ensure that robots have up-to-date positioning. However, care must be taken to not consume all of the available inter-robot communications bandwidth. A periodic rate of 4 hz was selected somewhat arbitrarily – it is fast enough to allow reasonably smooth real-time robot motions, while putting only a moderate strain on channel bandwidth. This update rate must be taken into consideration when designing servo loops based on neighbor position, as the robots can sometimes move faster then their neighbor's positions can be refreshed, which can cause instability even with modest gains.

The ISIS communications system runs at 250 kbps, but packetization and DC-balanced Manchester encoding reduce the throughput to 98.5 kbps for 64 bit packets, or 1538 packets/second. The 4 hz neighbor transmit cycles and the practical limit of the Aloha-like protocol limit communications to 192 packets/neighbor cycle. These 192 packets must be shared amongst all neighboring robots. For example, if you expect each robot to have 5 neighbors, then each robot can only transmit 38 packets per neighbor cycle.

Another way to limit the inter-robot communications usage is to reduce the number of neighbors each robot can detect. Since all neighbor communication packets use the same physical medium, robots cannot selectively ignore communications packets from specific neighbors. In order to limit the number of neighbors, the ISIS IR communication system has the ability to change the transmit power via software. This allows the software engineer to select a transmit power, based on the workspace the Swarm is using, to provide a desired expected number of neighbors. Currently, this transmit power is kept constant in each application, but more sophisticated software could vary the power level dynamically as a robot's local neighbor density varies.

Message Persistence

The ISIS communication system is reliable, but lossy. Often, the robots behavior is closely coupled to the received messages from neighboring robots. Lost packets can result in jerky motion and incorrect computations. In order to combat this, the most recent messages from each robot are buffered for a short time. New messages override the stored values. This "message persistence" provides some robustness to missed packets, but also preserves stale data from robots that have moved out of communication range. The number of cycles that messages are kept is a tunable parameter, with smaller values being more desirable. A persistence of four cycles produces acceptable results, based on subjective evaluation of the 100-robot swarm in many different environments and robot densities.

2.1.3 NeighborOps

The neighbor system populates a shared memory data structure with the most current neighbor status. User programs can read this data directly, or use the NeighborOps API for common operations. This collection of functions allows user programs to collect neighbors with specified characteristics into sets, then operate on the sets with standard operations. Functions can select neighbors based on range, robotID, gradient messages, or any application-specific data, then use standard set operators such as union and intersection to produce the desired set of neighbors.

NeighborOp Syntax

The general neighborOp function is:

```
setOut ⇐ nbrOp(setIn, condition),
```

where `setIn` is the set of neighbors to operate on, and `condition` specifies which neighbors to put into `setOut`. For example:

```
nbrSet ⇐ nbrOp(nbrSetAll, nbr.range < d),
```

will find all neighbors with range < d. `nbrSetAll` is system variable that contains all the current neighbors and `nbr` iterates over all elements in `setIn`. `nbrSetAll` can be shortened to `*`.

Output from neighborOps can be combined using set notation. For example:

```
nbrSet2 ⇐ nbrSet1 ∩ nbrOp(nbrSetAll, nbr.range < d),
```

will populate `nbrSet2` with the intersection of `nbrSet1` and all neighbors that are closer than d. There are also specialized functions and return sets sorted by common quantities. For example:

```
setOut ⇐ nbrOp-closestN(setIn, n),
```

or

```
setOut ⇐ nbrOp-furthestN(setIn, n),
```

will return the n closest or farthest neighbors from `setIn`. Many neighborOp functions return a single neighbor:

```
nbr ⇐ nbrOp-ID(setIn, robotID),
nbr ⇐ nbrOp-lowestID(setIn, condition),
nbr ⇐ nbrOp-closest(setIn, condition),
nbr ⇐ nbrOp-farthest(setIn, condition),
nbr ⇐ nbrOp-any(setIn, condition).
```

All of these will return either NULL (∅), or the neighbor with the feature in question. Appendix A1 contains the full C API for this system and some programming examples.

Chapter 3.
Gradient Message Propagation

Gradient communication messages provide a structured way to spread information throughout the swarm. [7] They can also perform useful distributed computations as they propagate, such as nominating leaders (section 3.1.2) or counting robots (section **Error! Reference source not found.**). In addition, the close relationship between network connectivity and physical location allows robots to use the gradients for long-range navigation (section 0).

We define:

n as the total number of robots.

$\mathbf{t_n}$ as the period of the neighbor transmit cycle.

g the number of different types of gradient messages in the current application. Different types of messages propagate independently. This will be explained in more detail below.

p as the persistence time for gradient messages. The most recent message of each type from each neighbor "persists" in the input buffer of the receiving robot for a set number of neighbor cycles or until a newer message replaces it. There is a separate buffer for each neighbor and each type of gradient.

$\mathbf{nbrs_i}$ as the set of neighbors that robot **i** can communicate with.

G as the graph created by combining all of the $\mathbf{nbrs_i}$ into a global data structure with robots as vertices and communication links as edges. Note that **G** can change every periodic neighbor transmit cycle.

There can be an arbitrary number of *types* of gradient messages, usually directly related to application functions. For example, an exploration application might have one gradient message type for scout robots, one type for robots acting as communication links, and one type for navigation to the charging stations. In all examples in this work, **g** is either constant or has an upper bound known at compile time. Each gradient type propagates independently.

Depending on the distributed computation being performed, different types of gradients are relayed slightly differently, but they all have some properties in common. Each gradient has at least one distinguished source robot, but there can be many more. In some applications, like the leader nomination example from section 3.1.2, every robot is a source. The gradient communications messages originate from the source robot and are relayed to its immediate neighbors. These neighbors become "one hop" robots, and they relay the gradient message to their neighbors who become "two hop" robots. This

process continues until the gradient reaches its maximum number of allowable hops or the edge of the network.

Robots relay gradient messages during every periodic neighbor transmit cycle. Therefore, the gradient tree is constantly being rebuilt, and is able to cope with the radical network topology changes that occur on a swarm of moving robots. The propagation time for a gradient message to disperse through the entire swarm with perfect communications is no greater than:

$$\mathbf{diam(G)} \cdot \mathbf{t_n}$$

where $\mathbf{diam(G)}$ is the diameter of the graph $\mathbf{G}$. This assumes that each hop will take the maximum time, $\mathbf{t_n}$. The expected hop latency between two unsynchronized robots is $\mathbf{t_p} = {}^{\mathbf{t_n}}/_{2}$. This is because a robot will receive a message uniformly at random within its neighbor cycle, hold it, then retransmit it at the end of the neighbor cycle. Because ISIS communications are lossy, there can be no upper bound on this propagation time, the worst case being when all packets are lost between the source and the other robots, resulting in no propagation.

Each robot maintains a global variable that stores current state for each gradient type. In addition, the most recent message of each type from each neighbor is stored in the neighbor information array. Because ISIS channels are lossy, all messages are buffered for a short time to make the system robust to small numbers of message losses. Each buffer is only one message deep, so any new communication will override the stored value. If no new messages of that type from that neighbor arrive in **p** cycles, the message buffer is cleared. A persistence value of 4 neighbor cycles works well in practice.

Gradient messages are implemented as structs and have two types of members, public and private:

type	The type of gradient message
sourceID	The robotID of the source of this gradient message.
senderID	The robotID of the sender of this gradient message.
hops	The number of times this message has been relayed. A value of ∞ indicates that this gradient has not been received during this cycle and is inactive. Inactive gradients are not relayed during the neighbor cycle
timeStamp	A time stamp used for the clean up functions described in section
data0-2	Three general purpose data bytes. These can be used by the user application or the processing function to spread information,

Table 1: Gradient Message Struct Public Members (transmitted to neighboring robots)

source	This is a Boolean flag that indicates weather or not this robot is a source of this gradient. It can only be set by the local robot.
timer	A timer to keep track of neighbor cycles.

Table 2: Gradient Message Struct Private Members (not transmitted to neighboring robots)

The public members are broadcast to all neighbors during the communication cycle. The private members are used by the transmitting robot for bookkeeping.

Using Gradients in Pseudocode

The pseudocode in Chapter 4 references gradient messages by using the gradient type as an index into an array of gradients. The messages on the robot running the pseudocode are stored in the array self.M[gType], and messages from neighbors are stored in nbr.M[gType], where nbr is a neighbor data structure.

To become a source for a gradient message, the syntax is one of:

```
gradientSource(self.M[gType], NORMAL),
gradientSource(self.M[gType], LATERALINHIBITION),
gradientSource(self.M[gType], COUNTING),
```

where gType is the type of gradient being sourced.

Gradients can be accessed using the neighbor ops functions. Comparisons using the variables self.M[gType], and nbr.M[gType], can be used in the standard neighborOp function:

```
nbrSet ⇐ nbrOp(nbr.M[gType].hops < self.M[gType].hops)
```

This line will find all neighbors that sent a gradient message of type gType with fewer hops than the message on the robot. This is a common operation, and will find parents on the gradient tree.

3.1 Gradient Propagation

Only one gradient message of each type is relayed during the neighbor cycle. This bounds the maximum number of gradient messages each robot will transmit per cycle to $\mathbf{g}$, and the maximum number any robot $\mathbf{i}$ will receive per cycle is $\mathbf{g} \cdot \mathbf{max}(\mathbf{nbrs_i})$. Most robots will receive multiple gradient messages of the same type and must select, combine, or otherwise process them in order to generate one message to be relayed. This job is performed by the processing function for that gradient type.

Each gradient type has one processing function, and the same function can be used for multiple gradient types. This function is called for each gradient type once every neighbor cycle. The syntax is f(M, m) where M is a pointer to a global gradient message variable and m is an array of all

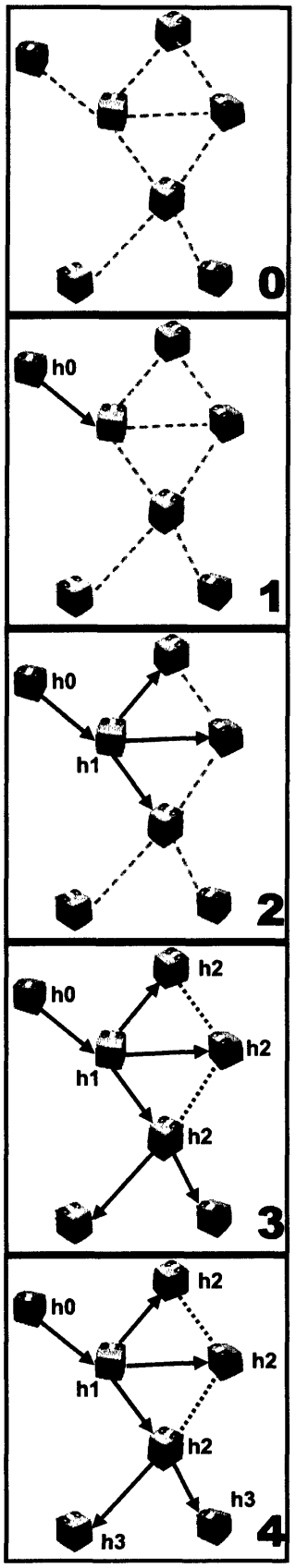


Figure 8: A communications gradient is formed as messages are relayed from robot to robot. The "h" numbers near each robot indicate how many hops the gradient message has traveled from the source.

the gradient messages of that type received during the current neighbor cycle. The function processes the received messages in **m**, then stores the results in M. After all the messages for each gradient type have been processed, the results stored in the global variables are transmitted. Care is taken in the system design to ensure that the gradient processing thread is mutually exclusive to all other threads that modify the gradient data. This prevents any thread from reading corrupt data. There are no constraints on the type of function that is used as a processing function; any function that processes the input and modifies the result can be used to process messages.

3.1.1 Normal Gradients

The most common processing function is processGradient. Its implementation is simple: robots that receive multiple gradient messages of the same type keep the one with the lowest hop count. This ensures that each robot keeps the message from a neighbor that is closer to the source than it is, eliminating cycles and creating a breadth-first tree on **G** that is rooted at the source robot.

processGradient(M, m)

```
1.  if M.source = TRUE
2.    M.hops ⇐ 0
3.    M.sourceID ⇐ MYROBOTID
4.  else
5.    M.hops ⇐ ∞
6.    for i ⇐ 1 to length(m)
7.      if (m[i].hops + 1) < M.hops
8.        M ⇐ m[i]
9.        M.hops ⇐ m[i].hops + 1
10.     endif
11.   endfor
12. endif
```

This is essentially the same algorithm presented in [Lynch 1996, pp 60]. The gradient messages "search" the graph, starting from the source. The M.source flag is set and cleared by the user application, usually in response to some behavioral event. Lines 1-2 initialize the source robot if needed. Lines 3-4 invalidate the current hop count of the gradient variable for non–source robots. Lines 5-9 find the message with the lowest hop count, taking care to add 1 to the values because the robot doing the computation is one hop away from all its neighbors. This message will be from the parent in the gradient tree.[5] At the conclusion of line 10, M.hops in robot i will contain one of two values: ∞, or $\min(\mathbf{nbrs_i}.\text{hops})+1$. If M.hops is ∞ this robot received no messages this cycle. Otherwise, the values in M will be copied from the best $\mathbf{m}_i$, with the hops in M adjusted. In particular, M.senderID is the parent of this robot in the tree.

[5] If there are multiple packets with the same hop count, then the robotID of the source and finally the robotID of the sender is used as a tiebreaker. This deterministic tiebreaking procedure reduces some chatter as robots will select the same neighbors to consider over multiple neighbor cycles, regardless of the ordering in the data structure.

Figure 8 is a step-by-step illustration of gradient propagation. The robot in the upper-left hand corner of the pictures is the source of the gradient. This message is broadcast omnidirectionally at the end of each neighbor cycle. Step 1 shows a robot receiving the message from the source. In step 2, this robot rebroadcasts the gradient message, and it is received by four robots – the three downstream robots and the source. However, when the source robot processes this gradient message, the hop count indicates that it traveled upstream, and the source discards it. This process continues for two more steps until the entire swarm has received the gradient.

Every robot that is in the same connected component as the source receives at least one gradient message from a neighbor that is closer to the source. Robots can use any of these parent neighbors to route communication packets to the source robot. Robots can also "route" themselves towards or away from the source by moving based on the positions of their parent or children neighbors. This physical routing is the basis for the `navigateGradient` behavior described in section 4.5.7.

There can be multiple sources of the same gradient type in the swarm. However, since the `processGradient` function will select the message with the fewest number of hops, a gradient with multiple sources will tessellate the swarm into groups. The process of selecting the source based on hop count is a discretized version of the closest-neighbors algorithm that produces a Voronoi tessellation of a normal graph. The top picture in Figure 9 shows an example of this tessellation in action, and the bottom picture shows the equivalent Voronoi tessellation using the sources as the vertices. Robots that are equidistant to multiple sources can randomly select a message from the set of closest sources, or use some ordering of the sources to select which message to relay. In practice, using robotIDs is

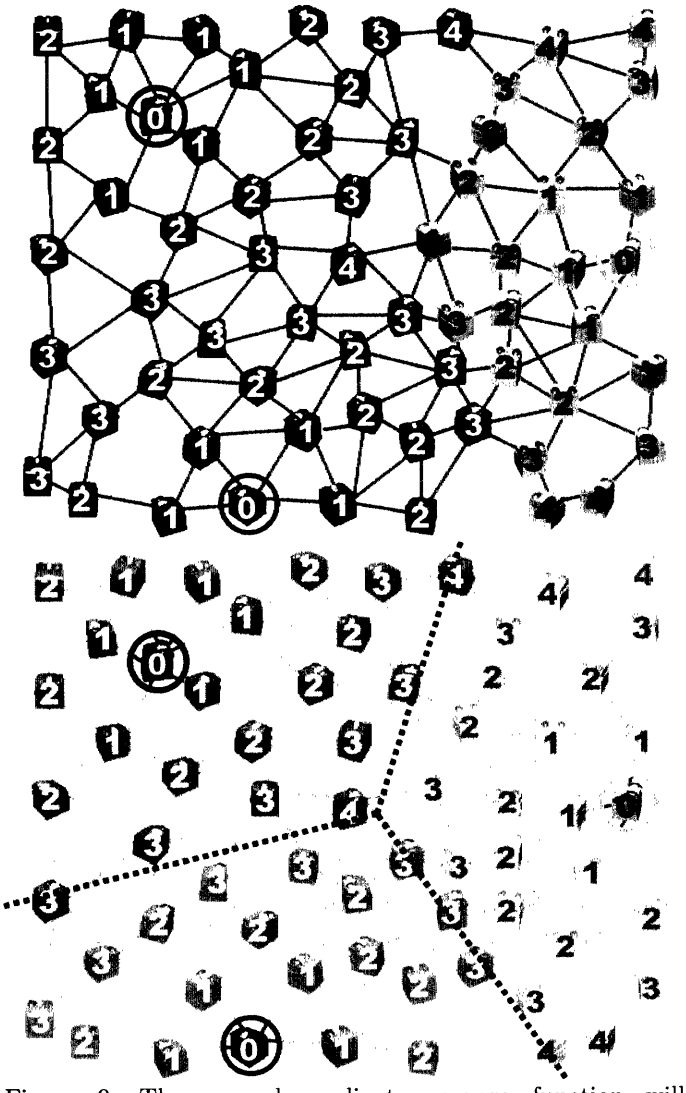


Figure 9: The normal gradient compare function will tessellate the swarm into Voronoi cells based on the number of hops each robot is from the source. This can be a convenient way to divide the robots into groups.

an effective tie breaker, and reduces dithering between two sources. In the example in Figure 9, the red source has the highest priority, then the green, followed by the yellow.

Normal Gradient Limitations

A classic distributed system has a static network and lossless communications. The Swarm does not have these properties, the robot network is very dynamic and has lossy communications. In addition, each message "persists" on the receiver for $\mathbf{p}$ neighbor cycles. This persistence is designed to allow robots to be more robust to missed packets, but if the network topology changes, robots will still retain copies of their old neighbors for $\mathbf{p} - \mathbf{1}$ cycles until their message persistence times out. Topology changes can cause problems similar to the situation shown in Figure 10, as a robot moves from one end of the network to the other. In this case, the communications can be disrupted for up to $\mathbf{p} \cdot \mathbf{diam(G)}$ cycles before messages from the correct source propagate back across the network. This type of failure can also be caused by communication links failing, then reconnecting. For example, if the highlighted robot in Figure 10 is sporadically connected to the bottom-left robot and the bottom-right robot.

Another problem is that if the source becomes inactive or is disconnected from the network, messages will "back propagate" from children to parents, ruining the tree structure of the breadth-first search.

The robots use the gradient tree structure for many different aspects of their behaviors, including communication and navigation. It is important for this structure to be constructed and deconstructed in an orderly fashion. The clean-up algorithms in section 3.2 address these requirements.

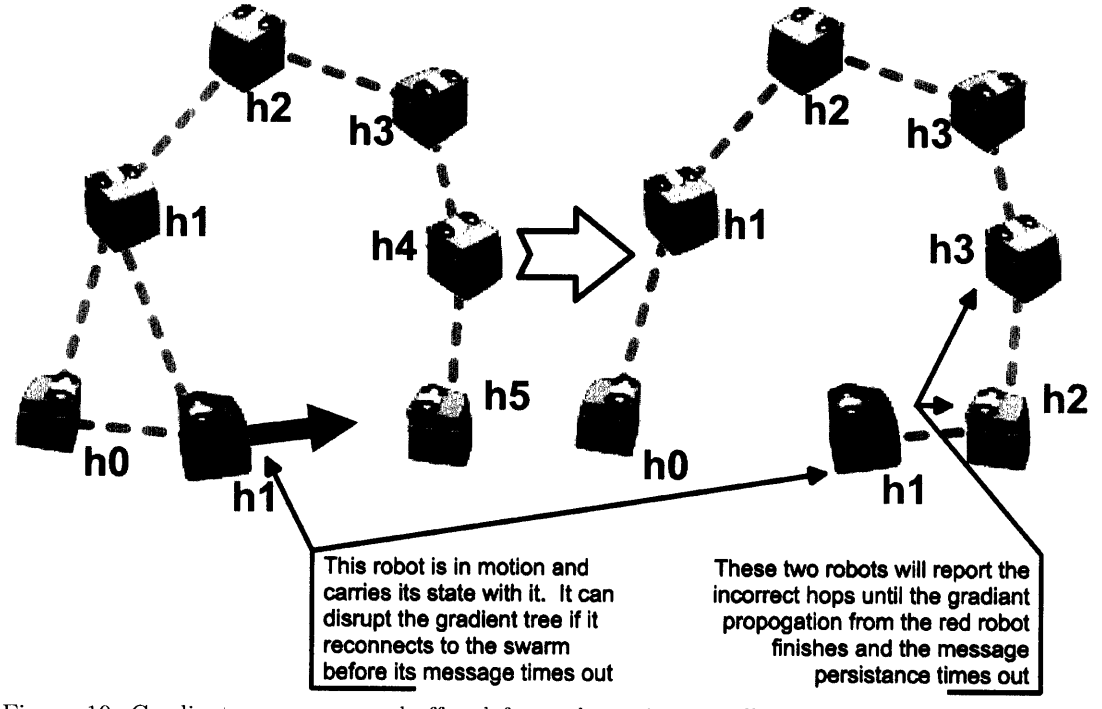
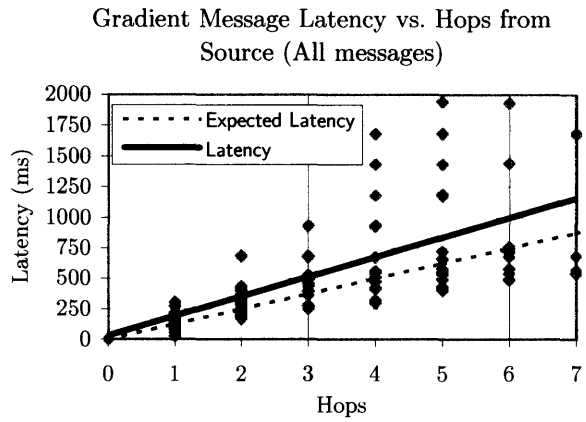


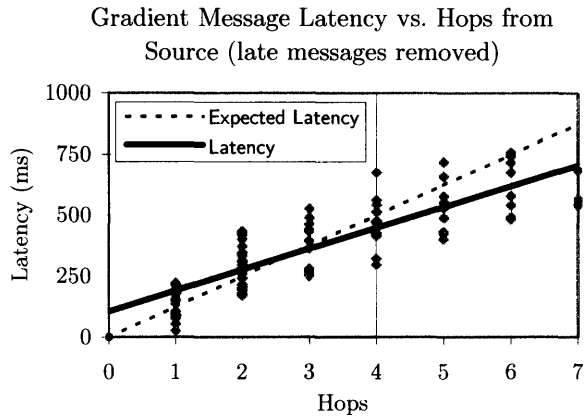
Figure 10: Gradient messages are buffered for a short time to allow the swarm to be robust to dropped packets. However, this buffering means that stored messages can be transferred between different parts of the swarm as robots move around. This can cause robots to compute the incorrect hops for a short time.

Experimental Results

The data in Figure 11 shows the arrival time of gradient messages at distant robots. Each time the message is relayed, it incurrs an additional expected latency if $\mathbf{t_p}$. Multiple runs were combined to produce this composite data set. The large data points are average values for each hop. ISIS is an infrared communication system, so links are line-of-sight and the diameter of **G** is dependent on the topography of the environment. With 100 robots the practical limit on diameter is about 40, resulting in an expected propagation time of about 7 seconds based on the data in Figure 11.



Average latency per hop = 172 ms



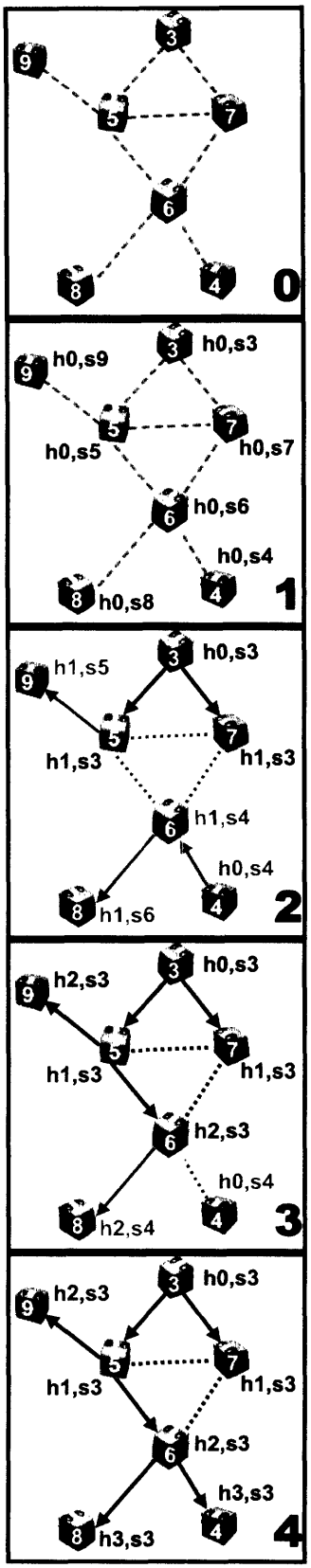
Average latency per hop = 125 ms

Figure 11: Gradient latency data from five propagation trials on a uniformly dispersed swarm. The size of the swarm ranged from 12-46, with an average size of 33 robots. The maximum diameter of the network is 7 hops. **Top:** Hop latency averaged 172 ms, 37% longer than $E(t_p)$ This is caused by lost packets that are only received after retransmission, and arrive late. **Bottom:** When late packets are removed from the data, the average hop latency is 125 ms, which is equal to $E(t_p)$.

3.1.2 Gradients with Lateral Inhibition

Symmetry breaking is a common task in distributed systems, as there are many algorithms require one robot to be distinguished from all others. For example, in the `followTheLeader` behavior from section 0, one robot must be the leader. The counting gradient in the next section requires one robot to tally the total count. Often it is not important exactly which robot becomes the leader, so long as there exists one robot that is, and all the other robots know that they are not.

A gradient with lateral inhibition can accomplish this task. It propagates in much the same way as a normal gradient, except the processing function gives preference to messages from the source with the lowest robotID, even if the message has traveled more hops. This means that the source of a message can be inhibited by another source with a lower robotID. Any globally unique property can be used instead of robotID. After the propagation is complete, the leader will be the one robot where the robotID of the source is equal to its own robotID.



processGradientLateralInhibition(M, m)

```
1.  if M.source = TRUE
2.    M.hops ⇐ 0
3.    M.sourceID ⇐ MYROBOTID
4.  else
5.    M.hops ⇐ ∞
6.    M.sourceID ⇐ ∞
7.  endif
8.  for i ⇐ 1 to length(m)
9.    if m[i].sourceID < M.sourceID
10.       M ⇐ m[i]
11.   else
12.       if (m[i].hops + 1) < M.hops
13.         M ⇐ m[i]
14.         M.hops ⇐ m[i].hops + 1
15.       endif
16.   endif
17. endfor
```

The algorithm is very similar to processGradient, with the addition of lines 9 and 10 which give the source robotID priority over hops. Line 6 initializes the global M. sourceID in case the previous source is no longer present. Note that the M.source flag does not necessarily report if this robot actually is a source or not, because it could be inhibited by another source with a lower robotID. The only way to know if you are an active source, and also the leader, is to compare M.sourceID to the local robotID. If they match, then this robot is the leader. However, if the two lowest IDs are on opposite ends of the network, it could take up to one gradient propagation time to determine this. Also note that until the gradient completely propagates, other robots will think that they are the leader. The multi-leader error will only exist until the gradient finishes propagating, but requires higher-level algorithms to be tolerant of this behavior. This software assures that there will be at least one leader, while a more sophisticated system of interlocking might be able to guarantee that there will be at most one leader.

The example in Figure 12 shows an example of leader nomination using a gradient with lateral inhibition. The hops from the source is indicated by h# and the source robotID is indicated by s#. The messages from robot 3 are drawn in dark green to make their propagation easier to distinguish from other messages, which are drawn in light

Figure 12: Sources of gradients with lateral inhibition can override other sources of the same type. The gradient message from the source with the lowest RobotID will be relayed. After one gradient propagation time, there will be only one source that is not inhibited.

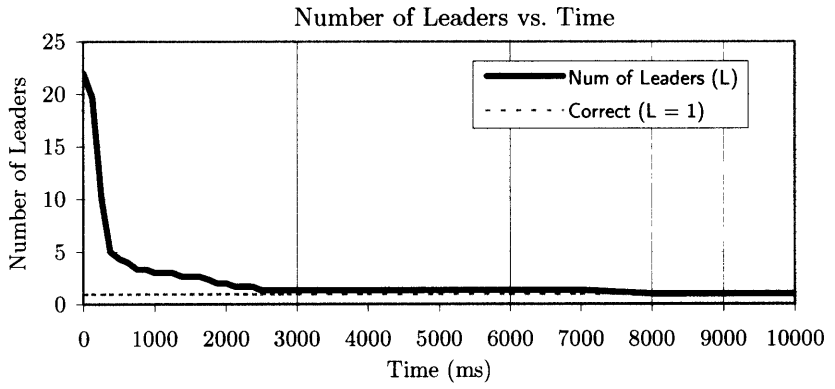


Figure 13: Gradients with lateral inhibition can be used to elect a leader among several robots. The graph above shows the number of leaders vs. time for several trials of this algorithm. It takes one propagation time for one leader to be elected and for all other robots to know it. In the data above, the leader is elected in an average of 4125 ms, which is very long compared to 1575 ms predicted by the data in Figure 11. This network was denser and had several turns, so packet collisions and bad links could have contributed to this error.

green. In Step 1, all the robots are sources, so the source robotID is the same as their own robotID and the message hops are all 0. Step 2 shows the gradient after one communications cycle, and the robots with lower IDs are starting to spread their influence. The propagation is complete in step 4, and robot 3 has inhibited all the other sources. It knows it is the leader, and the others know they are not. Note that if robot 3 is removed from the network, the gradient from robot 4 will then be allowed to spread after the messages from robot 3 are removed from the network. See the next section about gradient message clean-up.

Gradients with lateral inhibition do not tessellate the swarm like normal gradients. Instead, one robot's gradient propagates across the swarm, inhibiting all other sources of this type of gradient. This is not always desirable, but applications can limit the maximum number of hops these gradients can travel, or use a custom processing function to limit processing to a particular subset of robots, for example only those with a particular sensory input.

Experimental Results

Figure 13 shows the results of several trials of this algorithm. It takes one propagation time for one leader to be elected and for all other robots to know it. This average time to elect a leader is 4125 ms, which is very long compared to 1575 ms predicted by the data in Figure 11. This network was denser and had several turns, so packet collisions and bad links could have contributed to this error.

3.2 Gradient Clean-up

In a dynamic robot network, it is important for gradients to spread quickly and in a controlled fashion, but it is equally important for them to decay in a controlled fashion. Because the robots use the gradient trees for navigation, having them decay in an uncontrolled fashion can cause robots to move in unexpected directions. Consider Figure 14, which illustrates how a normal gradient that uses the `processGradient`

No Clean-up
Max Hops = 6

time	Source	1 Hop	2 Hops	3 Hops
0	0	1	2	3
1	2	1	2	3
3	2	3	2	3
4	4	3	4	3
5	4	5	4	5
6	6	5	6	5
7	6	-	6	-
8	-	-	-	-

Figure 14: Gradient message clean up is as important as propagation. The example network shown at top will be used for all discussions in this section. The chart shows how the hop counts decay after the source stops transmitting. Messages are only removed when they have been relayed for the maximum number of hops, in this case 6. This destroys the structure of the gradient tree because all robots will eventually relay a maximum-hop message. For large networks with high hop limits, this decay can take a long time.

processing function will decay if the source becomes inactive. Messages will be relayed from robot to robot until they reach the maximum number of allowed hops. Note how messages from the hop 1 robot back-propagated to the source after the source stopped transmitting. This ruins the structure of the gradient tree and for large networks with high hop limits, this kind of decay can take a long time, $\mathbf{p \cdot t_n \cdot diam(G)}$.

The ideal message clean-up is shown in Figure 15. This function takes the minimum number of neighbor cycles to clean up, and the total clean-up time is $\mathbf{t_n \cdot diam(G)}$.

There are two approaches to gradient clean-up described in this section, message clean-up, which transmits explicit clean-up messages, and time-stamp cleanup, which uses time stamps to eliminate back-propagation. Clean-up messages allow the gradient to be cleaned in the smallest possible time, but require the source to actively start the clean-up process, and require all other robots to relay clean-up messages. Timestamp clean-up will function even if the source is removed or disconnected unexpectedly, but takes longer to complete.

Ideal Clean-up

time	Source	1 Hop	2 Hops	3 Hops
0	0	1	2	3
1	-	1	2	3
3	-	-	2	3
4	-	-	-	3
5	-	-	-	-

Figure 15: The ideal gradient clean-up function would be able to remove a gradient in minimum time, without changing the hop values from the propagation. The minimum time is the same time required for the message to propagate. Gradients would be removed in an orderly fashion, in with robots removing messages in the reverse order they were received. The table above shows an ideal clean-up for the network of robots in Figure 14.

3.2.1 Message Clean-up

The source initiates a message clean-up when it transitions from being active to inactive. The clean-up message serves to tell other robots that this source is no longer active. Lets call this source robot **a**. When any robot receives a clean-up message, it enters the clean-up state for source **a** for **p + 1** neighbor cycles. While in this state, it removes all gradient messages that have originated from source a, and does not use them for computation or relay them to other neighbors. Instead, a clean-up message for source **a** is transmitted for **p + 1** neighbor cycles.

```
gradientCleanupMessage(M, m)
1.  if M.cleanUpTimer > 0
2.    removeMessages(m, M.cleanUpSourceID)
3.  endif
4.  for i ⇐ 1 to length(m)
5.    if m[i].type = CLEANUPMESSAGE
6.      removeMessages(m, m[i].sourceID)
7.      if M.cleanUpTimer = 0
8.        M.cleanUpTimer ⇐ CLEANUPCYCLES
9.        M.cleanUpSourceID ⇐ m[i].sourceID
10.     endif
11.   endif
12. endfor
13. if M.cleanUpTimer > 0
14.   queueCleanUpMessage(M)
15.   M.cleanUpTimer ⇐ M.cleanUpTimer - 1
16. endif
```

Line 1 checks to see if this robot is already in the clean up state. If so, the `removeMessages` function removes any messages in the input array **m** that are from the source stored in `M.cleanUpSourceID`. It also removes clean-up messages from

Message Clean-up
Max Hops = 6, Persistance = 3

time	Source	1 Hop	2 Hops	3 Hops
0	0	1	2	3
1	C	1	2	3
3	C	C	2	3
4	C	C	C	3
5	C	C	C	C
6	-	C	C	C
7	-	-	C	C
9	-	-	-	C
10	-	-	-	-

Figure 16: Clean-up messages can remove a gradient in the minimum time. Clean-up messages are transmitted for at least p + 1 cycles to reduce the possibility of back propagation. However, the cleanup messages consume bandwidth, and prevent the source from re-starting the gradient until the clean-up is complete. Also, if the source gets disconnected, or link-failure cycles occur in the network, the orderly cleanup can be compromised and degenerate into the max hop count cleanup from Figure 14.

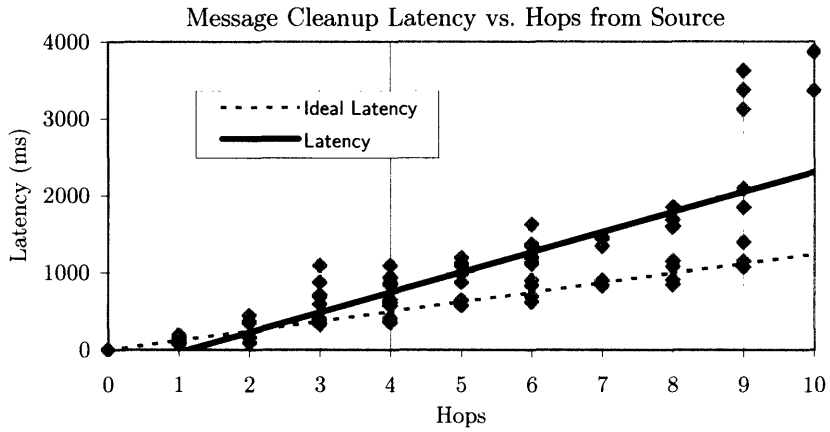


Figure 17: Message clean up data combined from five trials with 39 robots. The latency of 184 ms per hop is very similar to that of a gradient propagation, which had a latency of 172 ms/hop. The five data points on the upper-right of the graph illustrate the dangers of cycles that can persist for longer than the $\mathbf{p} + \mathbf{2}$ clean-up cycles. In this trial, the message clean-up still stopped the back-propagation of messages, but just barely.

neighboring robots that refer to that source. Lines 4-11 go through the input array looking for cleanup messages from other sources. If found, then all messages from that source are removed. If this robot is not already in the clean-up state, it is put into that state and the timer reset to CLEANUPCYCLES. This constant needs to be at least $\mathbf{p} + 1$ cycles. Larger constants provide more resistance to cycles from topology changes (Like the state error in Figure 10) and link failures, but prohibit the source from becoming active for a longer time. The minimum of $\mathbf{p} + 2$ worked well in practice. Lines 13-16 transmit a cleanup message if the robot is in clean-up state.

Note that clean-up messages from multiple sources cannot be relayed. Only the one source named in **M** is queued in the `queueCleanUpMessage(M)` function. This can introduce errors because multiple clean-up "wave fronts" will collide and interfere with each other, canceling parts of each other out. The only solution would be to relay clean-up messages from multiple sources each neighbor cycle. But since each robot can potentially be a source, and can begin cleanups asynchronously, this could lead to **n** messages needing to be relayed each neighbor cycle, which would consume all the available inter-robot communications bandwidth and violate our scalability design goals

In practice, this clean-up is of limited usefulness. Robots start and stop sourcing gradients often, and the system sends many clean-up messages, in some cases nearly as many as actual gradient messages. Communication errors are common, and complex environments can temporary disconnect large sections of the swarm for short periods of time. Since the source must actively initiate a clean-up, interruptions several hops away cannot be regulated, and the gradient tree structure quickly erodes on those subtrees.

Experimental Results

Figure 17 shows experimental data for message clean-up. This is the combined data from five separate trials. Message clean-up works well in stationary networks. Link failure cycles are rare, and the clean up is quick. On networks with moving robots, topology changes in conjunction with network failures can require clean-up times longer than $\mathbf{p} + 2$ to be robust. The time stamp clean-up in the next section address these issues.

3.2.2 Time-stamp Clean-up

Time-stamp clean up corrects the limitations of the message cleanup, but at the cost of a slower execution time. The source of a gradient maintains a time stamp variable that it increments each neighbor cycle. When the source transmits the gradient message, it puts its current time stamp in it. With perfect communications, each message a robot receives from that source will have a time-stamp value greater than the time-stamp of the last message received from that source. This invariant holds even when the message has been relayed through multiple robots, as there will be an ever-increasing chain of time-stamps leading back to the source. Robots can decide on the validity of any new message by comparing its time-stamp to that of the most recent received message. If the new time-stamp is greater than the most recent one then this message has traveled in a direct path from the source and should be kept. Otherwise, this message has been relayed from a sibling or child in the gradient tree and should be discarded.

```
gradientCleanupTimeStamp(M, m)
1.  if m.sourceID = MyRobotID
2.    M.timeStamp ⇐ M.timeStamp + 1
3.  endif
4.  for i ⇐ 1 to length(m)
5.    if ((m[i].sourceID = M.sourceID) and
          (m[i].hops ≥ M.hops) and
            (m[i].timeStamp ≤ M.timeStamp))
6.      removeMessage(m[i])
7.    endif
8.  endfor
```

Lines 1-3 update the timestamp if this robot is the source of the gradient. Line 5 looks for any message that...

1. ...is from the same source as the previous message of this type. Different sources of the same gradient type do not coordinate with each other and will have different time stamp values. Time stamps can only be compared when the messages come from the same source. This can cause problems if two robots stop sourcing at the same time, but the fix would violate scalability, as each robot would need to keep a copy of the last timestamp received from all possible sources of each gradient type. Since every robot can source a gradient, the upper bound on state would be $\mathbf{n} \cdot \mathbf{g}$, which is $\mathbf{O(n)}$.
2. ...has traveled as many or more hops from the source. Messages that have traveled fewer hops than the last message you have received should never be discarded.
3. ...has a time stamp that is the same or lower than the one from the previous message. Since the source is the only robot that can increment the timestamp, and it does so each neighbor cycle, all other timestamp values in the network will be less than the value on the source. Correct gradient propagation is from the source towards the leaves. Therefore, each robot should receive messages with an ever-increasing time stamp. If the timestamp is not greater than the one in the previous message, then this message is not from a parent in the gradient tree and should be discarded.

Time Stamp Clean-up
Max Hops = 6, Persistance = 3

time	Source	1 Hop	2 Hops	3 Hops
0	0	1	2	3
1	-	1	2	3
3	-	1	2	3
4	-	1	2	3
5	-	-	2	3
6	-	-	2	3
7	-	-	2	3
8	-	-	-	3
9	-	-	-	3
10	-	-	-	3
11	-	-	-	-

Figure 18: Time Stamp Cleanup uses a nondecreasing timestamp to eliminate back-propagation of gradients. It has the advantage of working if any part of the swarm becomes disconnected, but is p times slower than message clean-up.

If all these conditions are true, then this message has traveled backwards or laterally on the existing gradient tree and should be discarded. This will eliminate back-propagation if the source stops transmitting or becomes disconnected from the network. Figure 18 illustrates the number of cycles required to remove a message from the example network. The persistence in this example is 3 neighbor cycles. When the source stops transmitting, the timestamps will prevent the 1-hop robot from accepting any new messages, but it will keep its most recent message for **p** cycles before removing it. This process continues, until the gradient is completely removed from the network. Total clean-up takes no more than $\mathbf{p} \cdot \mathbf{diam(G)}$ neighbor cycles.

Bounded Time Stamps

While this algorithm will accomplish the goal of eliminating back-propagation, it uses unbounded time stamps. This is not practical on real systems, as even the largest integers will overflow. However, using a bounded time-stamp presents challenges because old time-stamp values must be reused. In order to know which messages to discard and which to keep, each individual robot must be able to determine the lower bound on the time stamp values that can exist in the network.

Because each gradient message can only travel a maximum number of hops before it is automatically removed, there can be a maximum difference in time-stamps of

$$\mathbf{p(M.hopsMax)}$$

between the most recent message transmitted by the source and the oldest message in the network. This happens when a message is received by a robot, but is not successfully retransmitted to any neighbors until the last persistence cycle, taking **p** cycles to travel one hop.

The most recent message on each robot contains the hops from the source and the timestamp of the source when that message was initially transmitted. This allows each robot to independently compute the timestamp of the oldest message that can still exist in the network from this source.

Eq 1 $\quad \mathbf{oldestTimeStamp = M.timeStamp - p(M.hopsMax - M.hops)}$

where **M** is the most recent message received from this source and **p** is the message persistence defined on page 30. Any messages with a time stamp lower than this value will have already been removed due to excessive hops, therefore values lower than this must be from new messages and should not be removed by the clean-up algorithm. We can now "wrap" the continuum of time-stamp values using modulo arithmetic and use values of finite size. In the code and explanations below, **R** is the base of the modulus.

gradientCleanupTimeStampMod(M, m)

```
1.  if m.sourceID = MyRobotID
2.    M.timeStamp ⇐ (M.timeStamp + 1) mod R
3.  endif
4.  for i ⇐ 1 to length(m)
5.    if ((m[i].sourceID = M.sourceID) and
          (m[i].hops ≥ M.hops) and
            (M.timeStamp ≥ m[i].timeStamp ≥ M.timeStamp – (M.hopsMax - M.hops)))
6.      removeMessage(m[i])
7.    endif
8.  endfor
```

In the Swarm, R is 256, which allows the time stamp to be encoded in one byte of data. Care must be taken to ensure that $\mathbf{p(M.maxHops) < R}$, or else *all* messages will be discarded. Another problem with using the modulus timestamp is that it could add a start-up delay to gradient processing. If a robot is disconnected from a source for a random amount of time, then reconnected, there is a probability

$$\mathbf{q = \frac{p(M.hopsMax)}{R}}$$

that the message that the robot last received from the source will fall into the range specified in line 5, the discard region. This robot would then discard all messages from this source for at most $\mathbf{p(M.hopsMax)}$ cycles, causing a delay before it started processing new messages. The number of neighbor cycles on which the robot will discard messages is not constant, but varies linearly within the discard region. The expected delay in cycles is given by:

$$\mathbf{E(delay) = q\left(\frac{p(M.\,max\ Hops)}{2}\right)}$$

substituting for q:

$$\mathbf{E(delay) = \frac{p^2(M.\,max\ Hops)^2}{2R}}$$

The practical limit on maxHops in the current swarm is about 40, giving a value for q of 62.5%, and an expected delay of 50 neighbor cycles, which is 12.5 seconds. This is somewhat long, but most experiments do not require packets to travel 40 hops. Reducing the max hops to 16 reduces the expected delay to 2 seconds, which is fine.

It takes at most p cycles for the time-stamp clean-up algorithm to propogate one hop. The maximum clean-up time is given by:

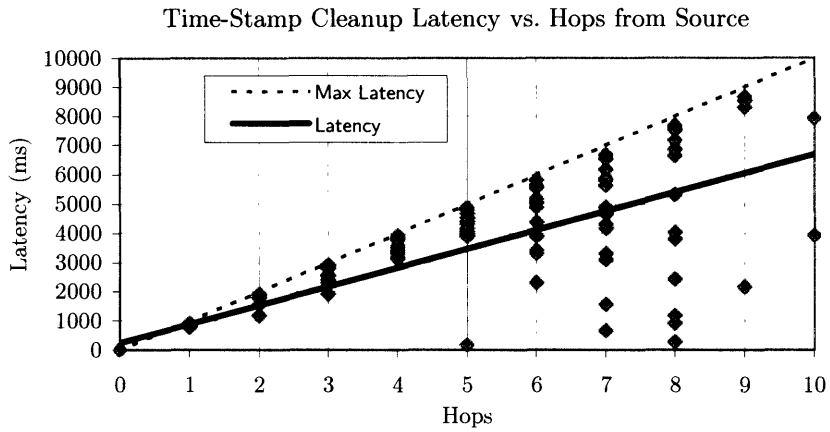


Figure 19: Data collected from time-stamp clean-up trials on a 39-robot swarm. The max latency line indicates the threshold between valid time-stamps and discard timestamps. In the swarm, **p** is 4 and t_n is 250 ms, which gives us a clean-up rate of 1 second/ hop. The clustering of data points underneath the max latency line shows how robots further away from the source remove their messages only after their parents have done so. The data in the lower right most likely is from network errors that cause those robots to lose their gradient message and record a spurious clean-up time.

$$\mathbf{t_{tscu} = p \cdot min(diam(G), M.hopsMax)}$$

The decay of messages is now orderly and the tree is deconstructed in the opposite order it was built. Robots that are dispersed remain in their positions. A useful side effect is that if the source loses contact with the network briefly, for example, when rounding a corner, the rest of the swarm starts an orderly clean up. When contact is reestablished, the active gradient "catches up" to the decaying gradient, typically in a few hops.

Experimental Results

The data in Figure 19 shows the results of five time-stamp cleanup trials on a 39-robot swarm. The key detail in the figure is that the distribution of clean-up times is completely below the max latency line. This solid performance makes time-stamp clean-up very useful in swarms of robots.

3.2.3 Combination Clean-up

It would be possible to combine message clean-up and time-stamp clean-up and get the fast expected clean-up time of message cleanup, with the security of time-stamp cleanup in the case of message failures.

3.3 Summary

The gradient messaging system forms the basis for almost all of the Swarms communications. Frequent retransmission allows the structure of the gradients to be robust to network topology changes, and clean-up algorithms preserve the structure of the gradient tree as the messages are removed from the network.

be

Chapter 4.
The Swarm Behavior Library

The ultimate goal for the Swarm project is to program group behaviors at the group level. For example, to explore the planet Mars you would want to type a program like this:

```
main(void)
{
   exploreMars();
}
```

We propose to break group behaviors into smaller behaviors that can be combined to achieve a larger goal. Continuing our example:

```
exploreMars(void)
{
   while(TRUE) {
      moveAwayFromTheLander();
      moveAwayFromOtherRobots();
      moveIntoUnexploredTerritory();
      if(fossilSensor == ACTIVE) {
            callForHelp(sensorRobot);
      }
      if(martianSensor == ACTIVE) {
            callForHelp(ambassadorRobot);
      }
   }
}
```

The statements within the while loop run concurrently, allowing the robot to respond to many different sensory conditions. This bottom-up solution provides some abstractions for the programmer, but she still needs to be aware of how different behaviors will interact, and the best ways to combine them to achieve the desired group performance. Understanding these relationships is the key to programming distributed systems. The behavior library presented in this chapter makes this task easier by providing reusable, scaleable behaviors that produce predictable group actions.

4.1 Behavior Operations

Behaviors are implemented as standard C functions that operate on a behaviorOutput data structure. The output of a behavior is a command for each output modality, contained within this data structure.

activationLevel	One of BEHAVIORINACTIVE, BEHAVIORACTIVE , or BEHAVIORDONE. These indicate the state of activation or completion of the behavior, and have the ordering of: BEHAVIORACTIVE > BEHAVIORDONE > BEHAVIORINACTIVE
translationalVelocity	This is the translational velocity that this behavior will request if it remains active
rotationalVelocity	This is the rotational velocity that this behavior will request if it remains active
LEDConfig	This controls the blinking of the status LEDs
speed	This is maximum speed (magnitude of rotV + transV) allowed for this behavior. This is set by the calling function, and is used to scale or limit the velocity outputs of the behavior.

Table 3: The behaviorOutput struct members.

This reduces the task of behavior arbitration to selecting which of these data structures will be used to control the robot. The architecture and philosophy is very similar to the subsumption architecture described by Brooks in [5], in which behavior operations are used instead of sensor data fusion.

The programmer's model is of multithreaded execution in which all the behaviors are running concurrently. Behaviors are simply C functions, which allows the use of a simple round-robin scheduler that periodically calls the main behavior function. Since this is cooperative multitasking, behavior functions must be non-blocking and terminate quickly.

The behavior system provides functions for operating on behavior outputs. Ultimately, the main behavior function produces a single behavior output that is passed to SwarmOS to drive the motors, behavior lights, and ISIS communication system. There are two primary behavior operators:

subsumeBehaviorOutputs

```
behOut ⇐ subsumeBehaviorOutputs(behLow, behHigh)
```

This operator compares the activation level of the higher priority behavior, **behHigh**, to that of the lower priority behavior, **behLow**. Table 3 shows the hierarchy of activation levels. If the higher priority behavior has a higher activation level than the lower priority behavior, its output will be returned by the function, otherwise, the output of the lower priority behavior will be returned.

sumBehaviorOutputs

```
behOut ⇐ sumBehaviorOutputs(beh1, beh2)
```

This operator combines `beh1` with `beh2`. The velocities are summed, the LEDOutputs are the union of the light patterns from `beh1` and `beh2`, and the higher activation level is passed to the output behavior.

4.2 Types of Behaviors

Figure 20 shows the static function-call tree of the behaviors presented below. They have been grouped into several categories, based on their goals and type of interactions with other robots. These categories are:

Demos

Demos are top-level behaviors or programs for demonstrating the swarm's capabilities to users, validating mission scenarios, and evaluating algorithm concepts.

Group Behaviors

These behaviors form the bulk of the behavior library. They are responsible for guiding the actions of a single active robot based on the positions and current state of all of its neighbors. The entire set of neighbors are the reference robots.

Pair Behaviors

Pair behaviors also direct the actions of a single active robot, but they use the position and current state of only one neighbor that is the reference robot. Some pair behaviors do not command any translational velocity. These are labeled as orientation behaviors on the graph in Figure 20.

Primitive Behaviors

These low-level behaviors do not interact with other robots at all. They provide low-level motion control and obstacle avoidance for an individual robot.

4.2.1 Functional Behavior Groupings

The Swarm Behavior Library can also be grouped based on functionality. Applications and Demos have been omitted because they are not library behaviors.

Motion
- moveArc
- moveStop
- moveForward
- moveByRemoteControl
- bumpMove

Orientation
- orientForOrbit
- orbitRobot
- orientToRobot
- matchHeadingToRobot
- followRobot

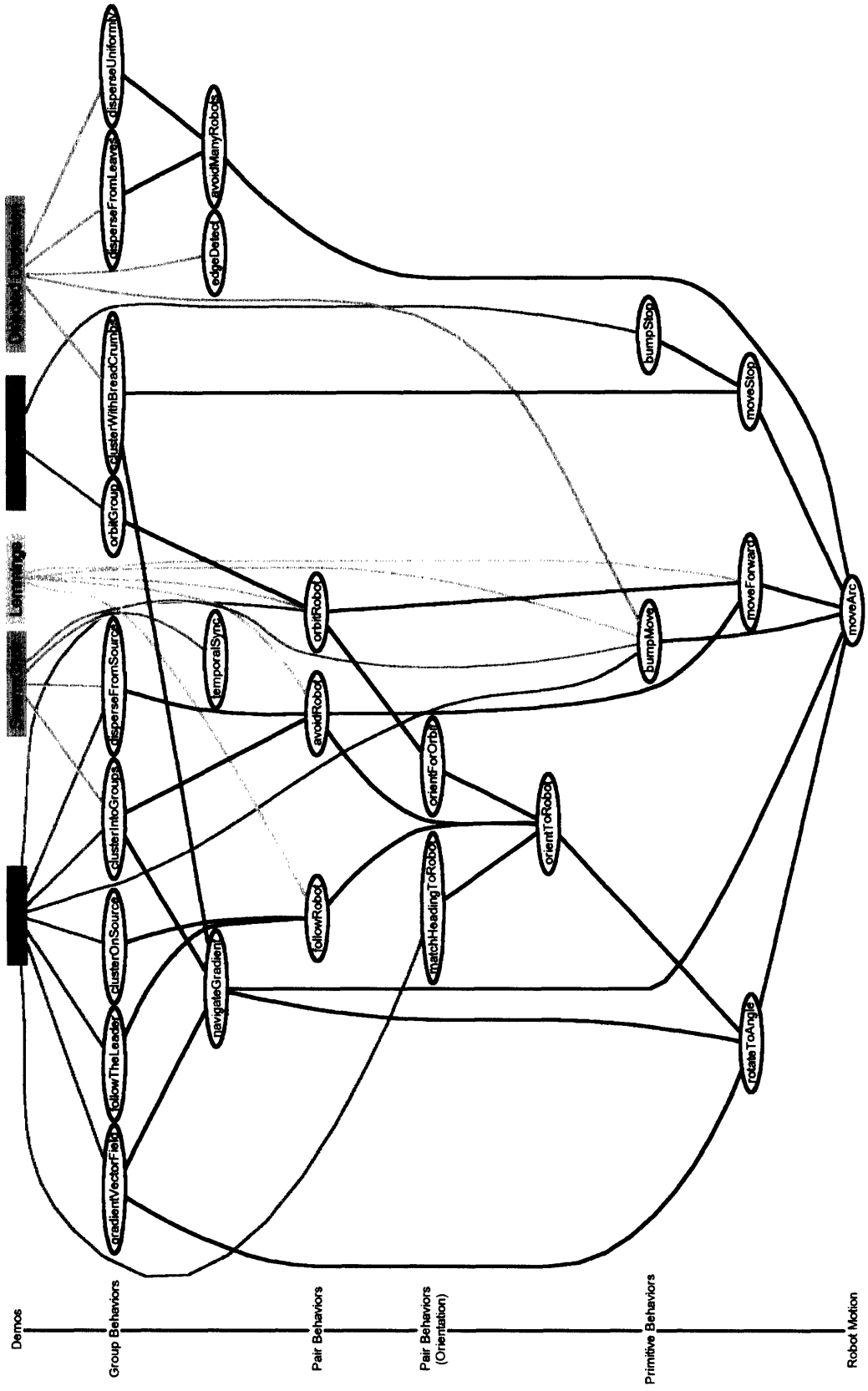
Navigation
- followTheLeader
- orbitGroup
- navigateGradient

Clustering
- clusterOnSource
- clusterWithBreadCrumbs
- clusterIntoGroups

Dispersion
- avoidRobot
- avoidManyRobots
- disperseFromSource
- disperseFromLeaves
- disperseUniformly

Utility
- detectEdges

Figure 20 (Next Page): This graph shows the static function call tree of the Swarm Behavior Library. Applications at the top, with more primitive behaviors appearing lower on the graph.


Demos
Group Behaviors
Pair Behaviors
Pair Behaviors (Orientation)
Primitive Behaviors
Robot Motion
gradientVectorField
followTheLeader
clusterOnSource
clusterIntoGroups
dispersefromSource
orbitGroup
clusterWithBreadCrumbs
dispersefromLeaves
dispenseUniformly
navigateGradient
temporalSync
followRobot
avoidRobot
orbitRobot
edgeDetect
avoidManyRobots
matchHeadingToRobot
orientForOrbit
orientToRobot
rotateToAngle
bumpMove
bumpStop
moveStop
moveForward
moveArc

4.2.2 Metrics

Two key metrics are used to quantify most behaviors: correctness and path efficiency. Correctness is a measure of how well a behavior is able to meet its specified goals. For most behaviors, these goals involve physical positioning of the robots. In these cases, the ratio of the shortest straight line path to the actual path is measured.

$$\mathbf{e} = \frac{\mathbf{shortestPath}}{\mathbf{actualPath}}$$

The resulting path efficiency is independent of speed and robot size, and can be used to compare performance across platforms, and even to biological systems.

4.2.3 Experimental Setup

All data was collected at the iRobot facility between January and May of 2004. Unfortunately, the Swarm's centralized data collection system was off-line for these experiments, so the apparatus shown in Figure 21 was used instead. Sanford bullet-tip flip-chart markers are preferred because of their resistance to bleed through to the other side of the paper. This allowed each sheet to be used twice, greatly reducing setup time for each experiment. A tape measure was used to measure linear distance, and a Trymeter "Mini Measure Maxx" rolling odometer was used to trace the paths of the robots. This is a very efficient measuring tool, and is precise enough to measure the finest detail that the robots produced.

The mounting location for the markers introduced two sources of error. The markers are attached to the rear of the robot, which causes them to draw an arc when the robot rotates in place. Also, the weight and friction of the magic markers often triggers the bump sensors. Attempts were made to eliminate paths from false bump responses whenever possible.

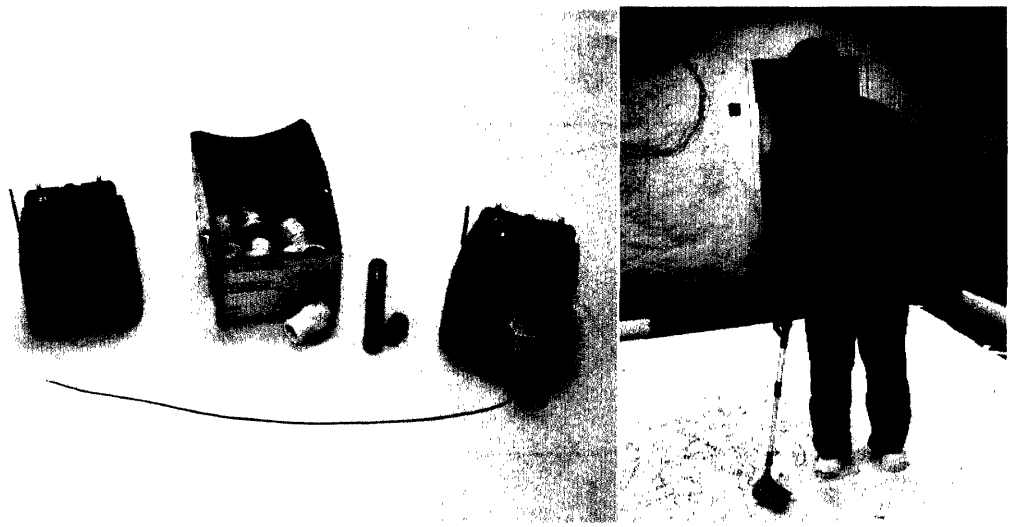


Figure 21: The state-of-the-art data collection hardware shown above used to measure path lengths on the swarm. **Left:** Each robot was instrumented with a magic marker. **Right:** The author measuring the path of a robot. (He is in a much better mood now)

4.3 Primitive Behaviors

These behaviors are at the lowest level of the behavior hierarchy. They do not rely on interactions with nearby robots in their implementation.

4.3.1 moveArc

Moves the robot using a given translational velocity and rotational velocity. This moves the robot such that its center follows an arc of radius

$$\mathbf{r} = \frac{\mathbf{b}}{2}\frac{\mathbf{v_t}}{\mathbf{v_r}}$$

where $\mathbf{b}$ is the lateral separation of the robot's wheels, $\mathbf{v_t}$ is the translational velocity and $\mathbf{v_r}$ is the rotational velocity. Positive translational velocities move the robot forward and positive rotational velocities rotate it clockwise.

moveArc(beh, t, r)

```
1. beh.translationalVelocity = t
2. beh.rotationalVelocity = r
```

4.3.2 moveStop and moveForward

These behaviors are syntactic sugar for moveArc, but are convenient to use, and allow the graph in Figure 20 to display behaviors that simply move forward or stop without combining them with the edges to moveArc.

moveStop(beh)

```
1. moveArc(beh, 0, 0)
```

moveForward(beh)

```
1. moveArc(beh, beh.speed, 0)
```

4.3.3 moveByRemoteControl

Moves the robot under remote control, usually for demos. This behavior is used to drive a single robot, while others operate autonomously around it. For example, the MegaDemo application from section 5.2 uses one robot driven via remote control to guide the behaviors of robots around it. Special hardware is required on the active robot to receive the radio control signals.

```
moveByRemoteControl(beh)
1. readRadioInputs(radio)
2. moveArc(beh, radio.translationalVelocity, radio.rotationalVelocity)
```

4.3.4 bumpMove

This is the primary obstacle avoidance behavior used by the swarm. When a robot collides with an obstacle, this behavior becomes active and moves the robot away from the obstacle. The action of this behavior is somewhat complex, and it is part of almost every swarm program. In order to simplify the descriptions of the algorithms presented we will assume that bumpMove is always running and that it is always successful in navigating robots away from obstacles.

```
bumpMove(beh)
1. [move robot away from obstacles]
```

4.4 Pair Behaviors

These behaviors are the simplest behaviors for interacting with neighboring robots. They move one robot, the active robot, in response to another, the reference robot. They build upon the primitive behaviors.

4.4.1 orientToRobot

The orientToRobot behavior rotates the active robot to a heading relative to the bearing the reference robot. The program specifies what bearing the active robot should maintain relative to the reference robot. Some examples are shown in Figure 22.

Spec

- Rotate so that the target robot is at the desired bearing and maintain this orientation.
- Minimize error and rotate to the goal orientation without overshoot as fast as possible.

```
orientToRobot(beh, nbr, bearing)
1. beh.translationalVelocity = 0
2. beh.rotationalVelocity = ko * (nbr.bearing - bearing)
```

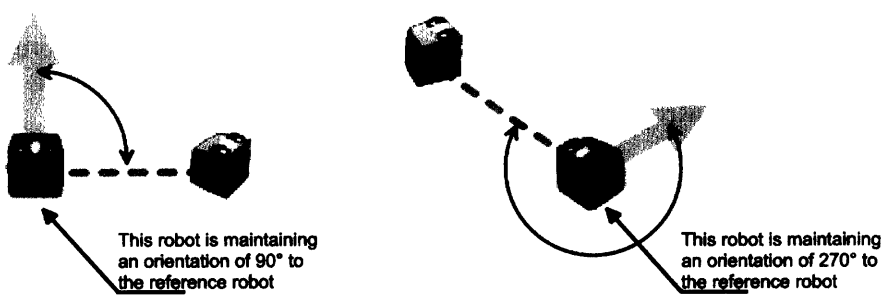


Figure 22: The orientToRobot behavior is a flexible way to orient an active robot with respect to a reference robot. The active robot rotates to the desired orientation relative to the reference.

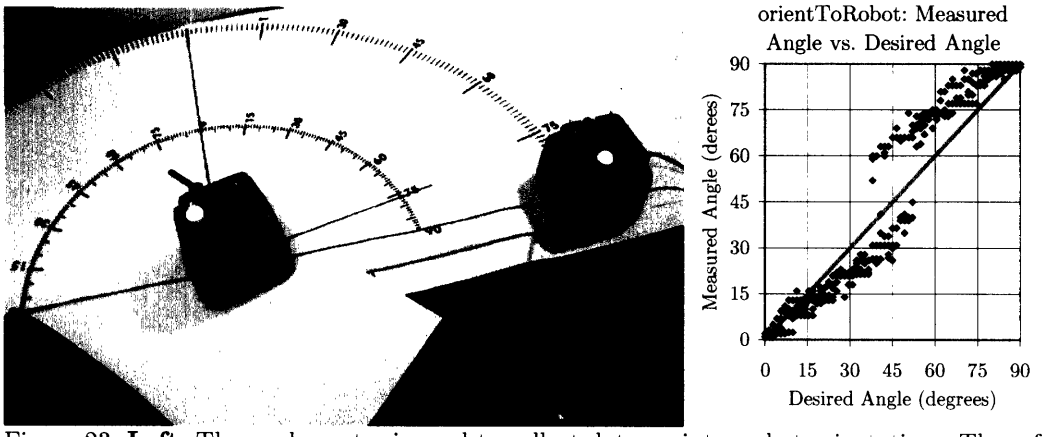


Figure 23: **Left:** The angleometer is used to collect data on inter-robot orientation. The reference robot has its red light flashing, and the active robot has its green light on. The wire protruding from the active robot sweeps across an angular scale. **Right:** The `orientToRobot` behavior has high resolution, but a discontinuity around 45°. Average error is 8.6°, with a deviation of 5.6° from the mean. Overall accuracy is 98% over 360°.

The active robot computes the bearing of the reference robot using the ISIS position system. The error between the actual position and desired is used as the input to a proportional control loop which rotates the active robot towards the reference. The gain term, ko, is selected to provide the fastest response without overshoot. The active robot need not face the reference robot, for example, the active robot could maintain an orientation such that the reference robot is located to its right hand side. This orientation is shows on the left hand side of Figure 22.

The gain of the control loop is currently limited by the 4 hz neighbor update rate and will become unstable if the active robot rotates too quickly and overshoots the reference. This error occurs when the active robot rotates past the desired final heading, then changes direction on the next ISIS cycle. The current motor control loop uses velocity control; a position control loop for rotation could reduce this type of feedback instability, and allow for control loops with higher gains.

Experimental Results

The reference and active robots were placed on an angleometer with a separation distance of 50 cm. The ISIS system has symmetry every 45°, so results obtained with input angles from 0-90 can be applied to the rest of the circle. The average error is 8.6° with a standard deviation of 5.6°, which gives a behavior correctness of 98% over the entire circle. This error is concentrated around 45°, which is one of the known limitations of the ISIS location system.

4.4.2 matchHeadingToRobot

The `matchHeadingToRobot` behavior uses both the bearing and orientation of the reference robot to direct the active robot to face in the same direction. It uses the `orientToRobot` behavior and the following equation:

$$\mathbf{b} = \mathbf{o} + 180°$$

where:

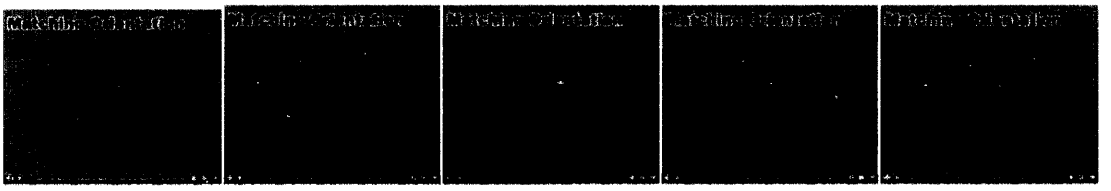


Figure 25: The pictures above show the **matchHeadingToRobot** behavior in action. The robot with the antenna is the reference robot, all other robots are active robots.

b is the desired bearing input to the **orientToRobot** Behavior in degrees

o is the orientation of the reference robot in degrees

In the picture in Figure 25, the robot in the center is under remote control, and all the other robots are matching their heading to it. This behavior is useful for making formations of robots. When a long-range ISIS beacon is the heading reference, as shown in Figure 29, this behavior can be used to move the robots along a global heading. The picture shows the robots moving "north", and as the beacon is rotated, they change their direction accordingly. The beacons look symmetrical, but they are directional and have a front. If the heading of the beacon is kept constant, the robots can use it like a compass to determine their absolute heading. Multiple beacons can be used to cover the entire workspace.

Spec

- Rotate such that orientation is the same as the target robot.
- Minimize error and move to the goal orientation as fast as possible without overshoot.

matchHeadingToRobot(beh, nbr)

```
1.  orientToRobot(beh, nbr.orientation + 180)
```

Line 1 uses the **orientToRobot** behavior to rotate the active robot, and is subject to the same dynamic constraints as that behavior. There are two types of orientation information available: direct orientation that has no additional latency, but only 45° of resolution, or reciprocal orientation that has 2° of resolution but incurs another periodic neighbor cycle lag (see section 1.2.2). In practice, this behavior does not require a fast response, so reciprocal orientation is used for its increased precision.

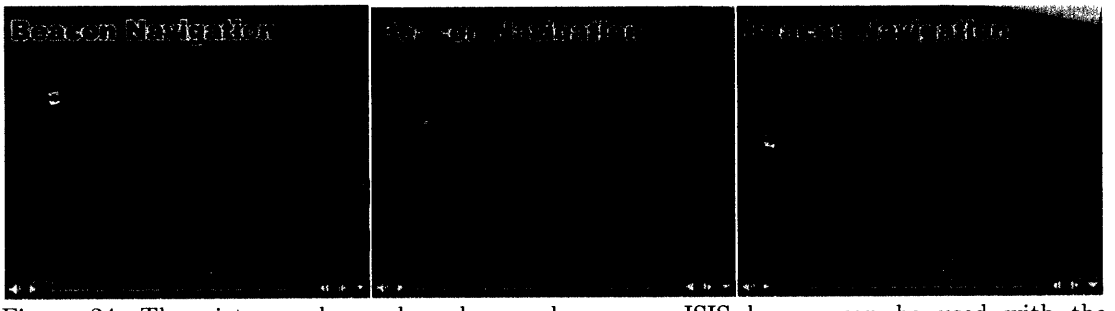
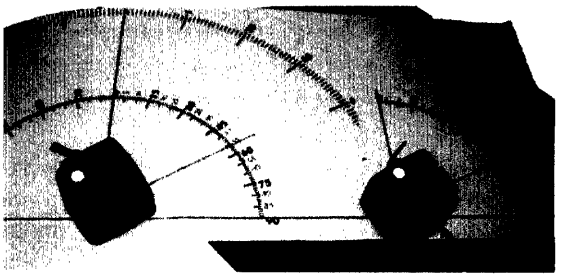


Figure 24: The pictures above show how a long-range ISIS beacon can be used with the **matchHeadingToRobot** behavior to guide the robots in along global heading. The robots are trying to move "north". As the beacon is rotated, all the robots change their direction accordingly. If the heading of the beacon is kept constant, the robots can use it like a compass to determine their absolute heading. Multiple beacons can be used to cover the entire workspace.



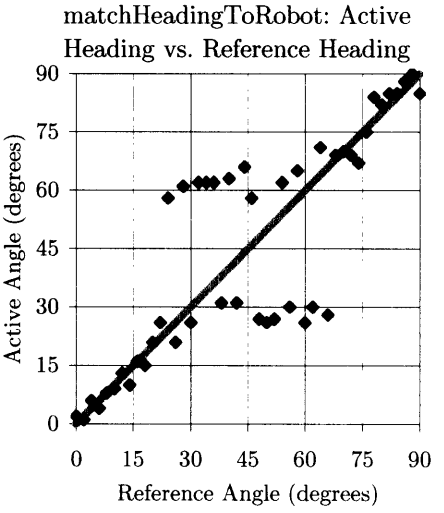


Figure 26: **Left:** The angleometer is used again to collect data on the `matchHeadingToRobot` behavior. The reference robot has its red light flashing, and the active robot has its green light on. Both reference and active robots have wire indicators. The reference robot is manually rotated to the input angle, which is selected at random to avoid having the active robot rotate in only one direction. **Right:** The `matchHeadingToRobot` behavior works well when ISIS works well, but fails when ISIS fails. Average error is 11.1°, with a deviation of 12.1° around the mean. Accuracy over the full 360° range is 97%.

Experimental Results

The errors in this behavior closely followed the errors in the `orientToRobot` behavior. In the areas where ISIS positioning works well, this behavior has very high accuracy.

4.4.3 followRobot

The `followRobot` behavior directs an active robot to follow a reference robot. This is a fundamental behavior and is used in many behaviors, including `clusterOnSource` and `followTheLeader`.

Spec

- Always be within a radius d of the reference robot.
- Always be facing the reference robot.
- The active robot should move along minimum shortest path to its final position at constant velocity. The final position is any pose that satisfies the above two constraints.

```
followRobot(beh, nbr, rd)
1.  orientToRobot(beh, nbr.bearing, 0)
2.  if nbr.range > rd
3.     beh.translationalVelocity = kf * (rd - nbr.range)
4.  endif
```

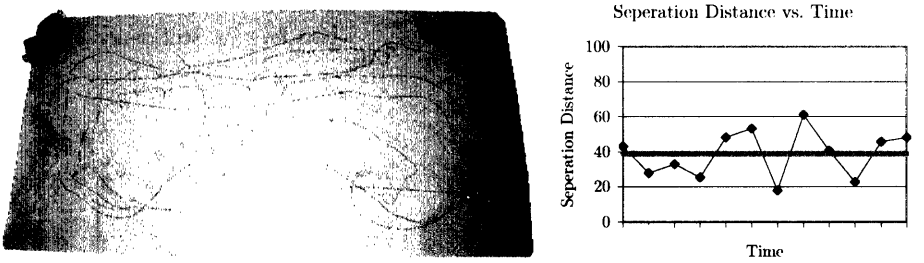


Figure 27: **Left:** The followRobot behavior produces the traces above. The red robot is the leader, and the green the follower. Each robot is programmed to "twitch" every two seconds so that their separation distances can be compared from the traces. **Right:** The commanded distance is 40 cm, and the average measured inter-robot separation is 39 cm, which yields a accuracy of 97% and a path efficiency of 114%. Both of these metrics were distorted somewhat because turns allow the follower to take "shortcuts" and round the corners on a shorter path than the leader.

Line 1 uses the orientToRobot behavior to keep the active robot facing the reference robot. No provision is made for cases where the orientation error is greater than 90 degrees, it is assumed that the orientToRobot behavior will respond fast enough to make this situation short-lived. Lines 2-4 form a control loop that is responsible for maintaining the desired range to the reference robot. It can be expressed with the following piecewise function:

$$v_t = \begin{cases} k_f(r_d - r_a) & (r_a \geq r_d) \\ 0 & (r_a < r_d) \end{cases}$$

where:

v_t (beh.translationalVelocity) is the active robot's commanded velocity,

r_d (rd) is the desired range to the reference robot,

r_a (nbr.range) is the actual range to the reference robot,

k_f (kf) is the control loop proportional gain constant.

The top equation generates a proportional control loop to have the active robot maintain the desired range from the reference robot. The bottom equation prevents the active robot from moving in reverse if it is too close to the leader. Although this discontinuity causes problems in the control loop, having robots moving in reverse while surrounded by other robots can be more problematic.

The proportional control loop will leave a steady-state error in tracking the reference robot that depends on how fast it is moving. If the reference robot is moving at constant velocity, the active robot will follow with an error of:

$$e = r_a - r_d = \frac{v_{ref}}{k}$$

To combat this, the controller has an integral term to reduce this steady-state error. Piecewise control logic clears the integrator state when the active robot gets within r_d of the reference.

Experimental Results

Figure 48 in the `followTheLeader` behavior section shows video clips of this behavior in action. Figure 27 shows some traces from this behavior. The commanded distance is 40 cm, and the average measured inter-robot separation is 39 cm, which yields an accuracy of 97%. Path efficiency is 114%, but turns allow the follower to take "shortcuts" and round the corners on a shorter path than the leader. This artificially increases path efficiency and adds error to the separation distance measurement. However, the follower never got separated from the leader. Using the "twitch" technique caused problems in measurement, especially around turns where the twitches could be easily obscured in the motions of the robot. A calibrated video imaging system would provide more accurate results

4.4.4 avoidRobot

The `avoidRobot` behavior directs an active robot to move away from a reference robot. This behavior is used for dispersion.

Spec

- The active robot should always be further than distance **d** from the reference robot
- The active robot should move along the shortest path to its final position at constant velocity. The final position is any one that satisfies the above constraint.

```
avoidRobot(beh, nbr, d)
1.  beh1 ⇐ EMPTYBEH
2.  beh2 ⇐ EMPTYBEH
3.  if nbr.range < d
4.    orientToRobot(beh1, nbr.bearing, 180)
5.    moveForward(beh2)
6.    sumBehaviors(beh, beh1, beh2)
7.  endif
```

This behavior is very similar to the `followRobot` behavior. Line 3 activates the

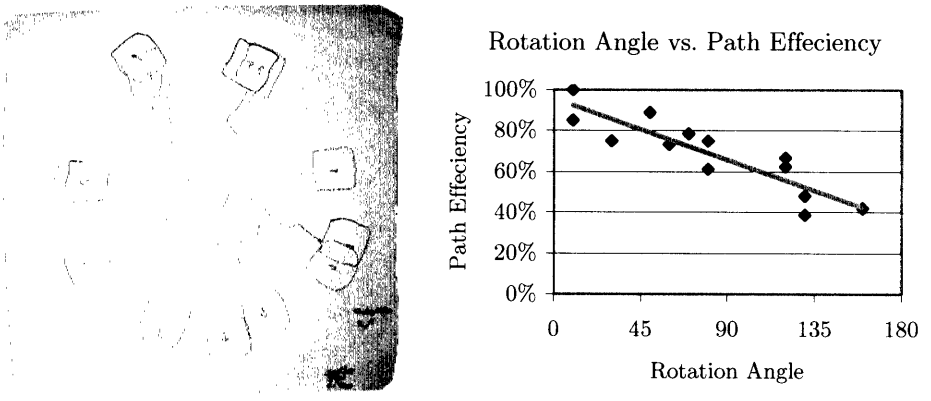


Figure 28: **Left:** The `avoidRobot` behavior produces the traces shown. The red robots were started facing upper-left, and the green traces are from robots released facing the upper-right. The summation of `orientToRobot` and `moveForward` produces the "J"-shaped traces shown. **Right:** The data shows path efficiency declines as the rotation angle increases. Average path efficiency is 69%, and average distance accuracy is 93%.

behavior if the range condition is violated. Lines 4-5 use the `orientToRobot` behavior to face the active robot away from the reference and the `moveForward` behavior to move it forward. The two behaviors are combined in line 6, the net result being motion away from the reference.

Experimental Results

The behavior moves the active robot away from the reference. The path efficiency varies depending on the angle the active robot has to turn to get away from the reference. The worst case scenario occurs when the active robot is facing the reference robot and begins to avoid it. The active robot will move in a "J" pattern as the `orientToRobot` behavior turns the robot while `moveForward` is active. This can be seem in Figure 28.

This error can be reduced by implementing the behavior such that the active robot only moves forward when it is facing away from the reference. However, it would produce jerky robot motion and require a tuning parameter for the allowable bearings to activate `moveForward`.

4.4.5 orientForOrbit

This behavior orients an active robot with respect to a reference robot such that if the active robot were moving forward, it would move in a circular path around the reference robot.

Spec

- Orient the active robot to move away from the reference robot if they are too close, to move towards the reference robot if they are too far, and transition smoothly between these two directions for an intermediate region.

```
orientForOrbit(beh, nbr, rd, orbitDir)
1.  if nbr.range > rd + c
2.    dir = 0
3.  else if nbr.range < rd - c
4.    dir = 180
5.  else
6.    if orbitDir = CLOCKWISE
7.      dir = 180 – nbr.range * ( 90 / c )
8.    else
9.      dir = 180 + nbr.range * ( 90 / c )
10.   endif
11. endif
12. orientToRobot(beh, nbr.bearing, dir)
```

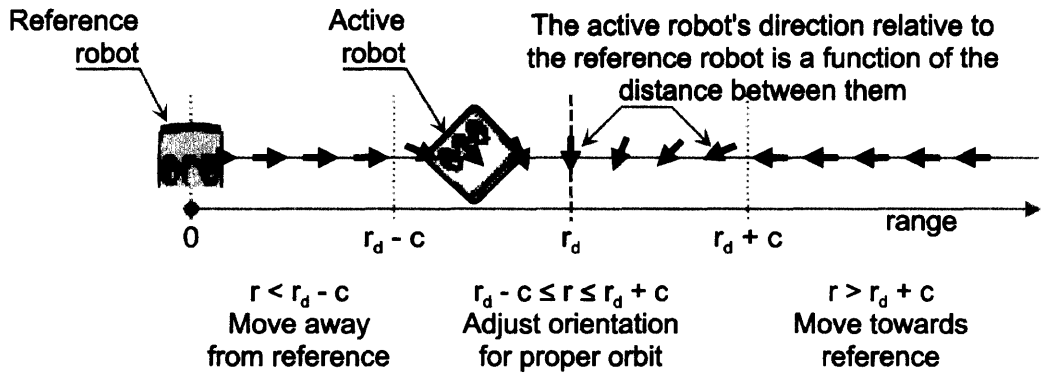


Figure 29: The `orientForOrbit` behavior orients an active robot relative to a reference robot. If the active robot is too close to the reference, the active robot turns away. If the active robot is too far from the reference, it turns to face the reference. The orientation is used by the `orbitRobot` behavior to guide the active robot in a circular path around the reference robot. The arrows indicate the orientation of the active robot as a function of range.

The orientation direction from the `orientForOrbit` behavior is given by:

Eq 2
$$\mathbf{dir} = \begin{cases} 0^\circ & (\mathbf{r} > \mathbf{r_d} + \mathbf{c}) \\ 180^\circ - \dfrac{90^\circ}{\mathbf{c}}\mathbf{r} & (\mathbf{r_d} - \mathbf{c} \le \mathbf{r} \le \mathbf{r_d} + \mathbf{c}) \\ 180^\circ & (\mathbf{r} < \mathbf{r_d} - \mathbf{c}) \end{cases}$$

for a clockwise orbit and

Eq 3
$$\mathbf{dir} = \begin{cases} 0^\circ & (\mathbf{r} > \mathbf{r_d} + \mathbf{c}) \\ 180^\circ + \dfrac{90^\circ}{\mathbf{c}}\mathbf{r} & (\mathbf{r_d} - \mathbf{c} \le \mathbf{r} \le \mathbf{r_d} + \mathbf{c}) \\ 180^\circ & (\mathbf{r} < \mathbf{r_d} - \mathbf{c}) \end{cases}$$

for a counterclockwise orbit, where:

dir (`dir`) is the desired orientation of the active robot relative to the reference robot,

$\mathbf{r_d}$ (`rd`) is the desired orbit radius,

r (`nbr.range`)is the actual radius (range) to the reference robot,

c (`c`) is a constant that determines the width of the transition region.

Most of the work is done in lines 6-10, which represent the middle cases in both equations. When the distance between the active robot and the reference robot is greater than the desired radius, the active robot turns towards the reference. If the distance is less than the orbit radius, the active robot turns away. Lines 1-5 limit the active robot to pointing directly at the reference robot or pointing directly away. Figure 29 shows the orientation for a clockwise orbit and the three different range zones corresponding to the three cases in Eq 2. Selection of the constant c affects the size of the transition zone between pointing towards and pointing away from the reference robot.

Experimental Results

Figure 30 shows the results of testing the behavior. Qualitative performance is good, the active robot will orient in such a way as to move away from the reference

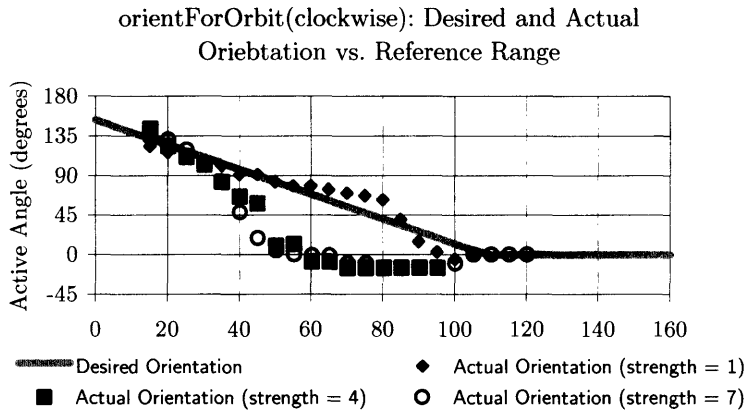


Figure 30: The **orientForOrbit** behavior attempts to orient the robot in such a way that if it were moving forward, it would move in a circle around the reference robot. The gray line shows the desired orientation for an orbit of 45 cm, with 90° (reference robot to the right) occurring at 45 cm. The behavior works well at close ranges and around the orbit radius, but the performance is poor further away. Fortunately, these types of errors will not adversely effect the qualitative performance of the function, robots that are further away than the desired radius will orient themselves to face the reference robot. The error is $|\text{desired} - \text{actual}|$ and the average is 28.5°, with a standard deviation of 26.3°. Accuracy over the full 180° range is 84%.

robot if too close, and move towards the reference if too far. Quantitative performance beyond 50 cm was far from the ideal orientation. As the range approaches the limits of the ISIS system, the range information has more errors. In addition, this behavior uses **orientToRobot**, and will have errors from that behavior as well.

4.4.6 orbitRobot

This behavior guides one robot around another in a circular path. The moving robot is the active robot, and the robot at the center of the circular path is the reference robot. This behavior is used in the **orbitGroup** behavior, and for inter-robot positioning.

Spec

- Move the active robot in a circular path of radius d around the reference robot.

orbitRobot(beh, nbr, d, orbitDir)

1. beh1 ⇐ EMPTYBEH
2. beh2 ⇐ EMPTYBEH
3. orientForOrbit(beh1, nbr, d, orbitDir)
4. moveForward(beh2)
5. sumBehaviors(beh, beh1, beh2)

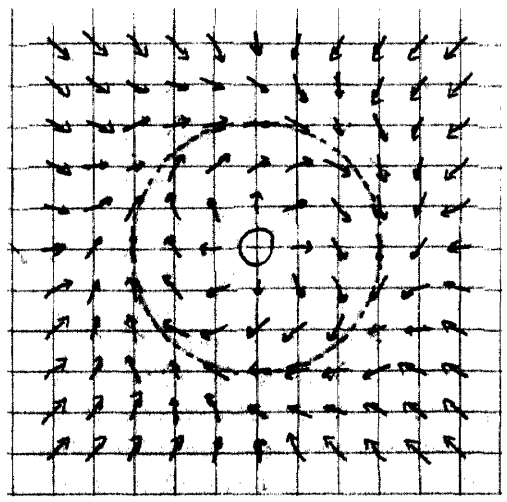


Figure 31: This is a sketch of the **orbitRobot** vector field. This vector field is the sum of the velocity outputs of **orientForOrbit** + **moveForward** produce a velocity vector field with a stable limit cycle at a radius **d** from the reference robot. Robots that move along this field will orbit the reference in approximately a circular path..

Figure 32: These video clips show a group of robots orbiting a reference robot. The reference robot is the one with the tall black antenna.

Lines 1-2 initialize temporary storage. Lines 3-4 run the two component behaviors, and line 5 combines their outputs into one resultant behavior. The `orientForOrbit` behavior combined with the `moveForward` behavior produce the two-dimensional velocity vector field centered around the reference robot shown in Figure 31. This field has a circular stable limit cycle, the circular path of radius r around the reference robot. A robot placed anywhere in this vector field will converge to this limit cycle.

This implementation has the advantage that the orbiting robots need to maintain no state about their orbit, only the last measurement of range. If the reference robot moves, this behavior will gracefully degrade into behaviors very similar to `followRobot` or `avoidRobot`, depending on the new range. This gives the behaviors considerable robustness to interference from other behaviors, obstacles, and communication errors. It is very similar in spirit to the vehicles proposed by Braitenburg. [16]

Experimental Results

Figure 32 shows some images from a video clip, and figure Figure 33 shows the traces left behind an orbiting robot. The stable limit cycle to attract a robot from any x-y position relative to the reference is evident in the middle picture of Figure 33. In addition, the active robot can be started at any orientation relative to the reference, and will rotate towards the limit cycle, as shown in the right-hand image. Figure 34 shows a

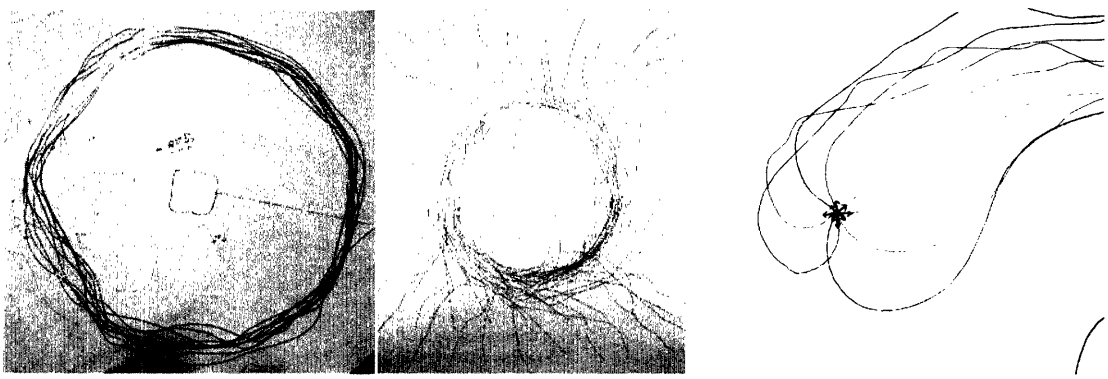


Figure 33: These traces show some results of `orbitRobot` behavior. All orbits are clockwise. **Left:** A steady-state orbit is quite stable, but shows some systematic errors at certain locations. **Middle:** The `orbitRobot` behavior uses a stable limit cycle to guide the active robot around the reference. **Right:** This detail view shows the path of an active robot when released facing the cardinal and intercardinal directions. The reference robot's position is in the lower right corner. The active robot quickly turns around and follows the trajectories back to the limit cycle.

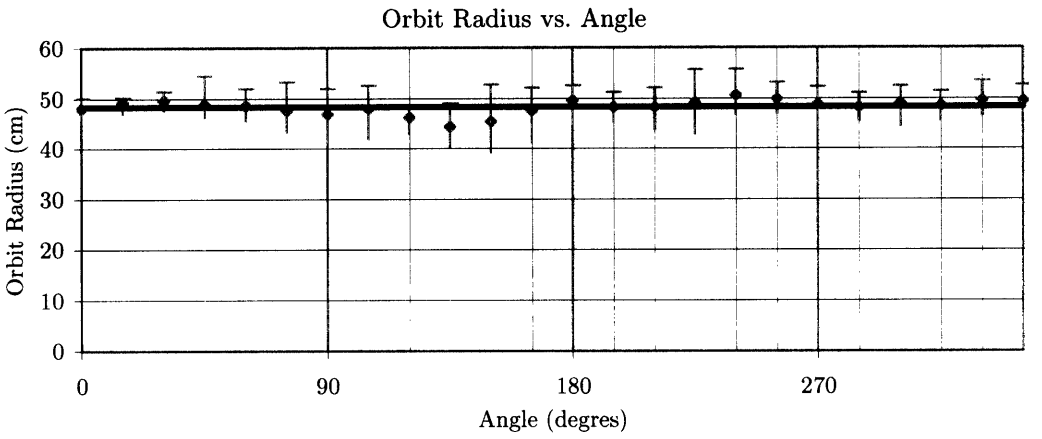


Figure 34: The orbit radius vs. angle is shown above. The error bars denote minimum and maximum radii. The commanded radius is 45 cm, and the average measured radius is 48 cm. The slight sinusoidal error in the data is most likely the result of not locating the exact center of the reference robot after it was removed. Over the entire sample, the deviation from the mean is 2.8 cm and the accuracy is 93%.

plot of the orbit radius vs. the angle. Overall, the orbiting is quite good, with an over all accuracy of 93%. The stability given by the vector field approach makes it possible to use this behavior in unstructured environments and with many robots nearby.

4.5 Group Behaviors

4.5.1 avoidManyRobots

The `avoidManyRobots` behavior directs a active robot to move away from a set of reference robots. This behavior is used in the `disperseFromSource` and `disperseFromLeaves` behaviors in the following sections.

Spec

- The active robot should always be further than distance **d** from all reference robots
- The active robot should move along shortest path to its final position at constant velocity. The final position is any one that satisfies the above constraint.

First we define a helper function, `computeAverageBearing`:

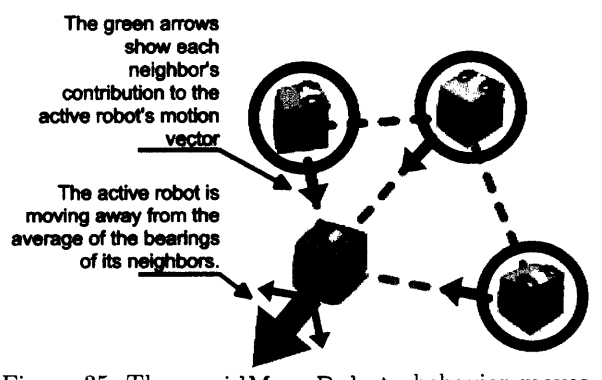


Figure 35: The `avoidManyRobots` behavior moves an active robot away from the average direction of the reference robots.

computeAverageBearing(nbrSet) returns b

```
for i ⇐ 1 to length(nbrSet)
  v ⇐ v + unitVector(nbrSet[i].bearing)
endfor
b ⇐ arctan(v)
```

The variable v is a vector, and the function unitVector(θ) takes an angle as an input and returns a unit vector rotated to the given angle. This function returns the angle of that resultant vector.

avoidManyRobots(beh, nbrSet, d)

```
1. beh1 ⇐ EMPTYBEH
2. beh2 ⇐ EMPTYBEH
3. nbrSet ⇐ nbrOp(*, nbr.range < d)
4. if nbrSet ≠ ∅
5.   b ⇐ computeAverageBearing(nbrSet)
6.   rotateToAngle(beh1)
7.   moveForward(beh2)
8.   sumBehaviors(beh, beh1, beh2)
9. endif
```

This behavior is very similar to the avoidRobot behavior. Line 3 discards all neighbors that are further than d from the active robot. Line 4 computes the average bearing to this set of neighbors. Lines 5-7 rotate the active robot to the desired heading while moving it forward.

This behavior has the same worst-case scenario as avoidRobot, when the active robot is facing the reference robots and begins to avoid them. The active robot will move in a "j" pattern as the rotateToAngle behavior turns the robot while moveForward is active.

Experimental Results

Figure 36 shows the data collected from this behavior. The optimal final position can be computed based on the positions of the references and the starting location of the active robot. The avoidManyRobots behavior has a final position

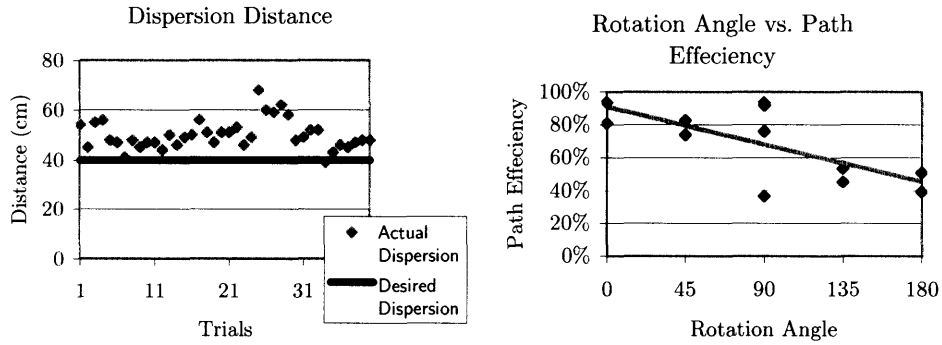


Figure 36: The avoidManyRobots behavior has a final position accuracy of 76% and a path efficiency of 68%. **Left:** The avoidManyRobots behavior achieves the desired separation distance of 40 cm. **Right:** Like avoidRobot, the path efficiency decreases as the angle the active robot has to rotate to move away fro the references increases.

accuracy of 76% and a path efficiency of 68%. Like `avoidRobot`, the path efficiency decreases as the angle the active robot has to rotate to move away from the references increases.

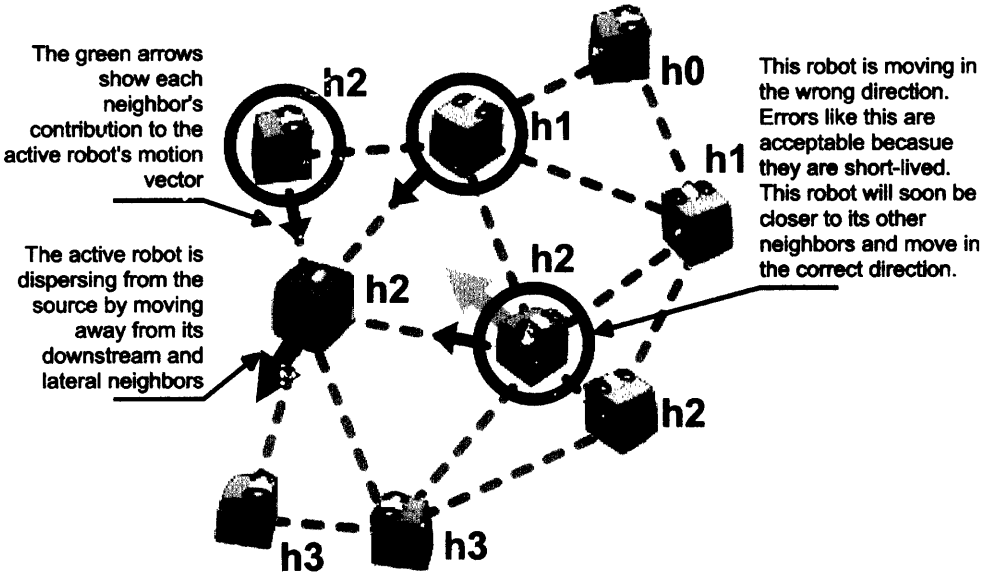


Figure 37: These robots are dispersing from the source of a gradient communication using the `disperseFromSource` behavior. The active robot moves away from all neighbors that are fewer or equal hops from the source.

4.5.2 disperseFromSource

Dispersion is a core behavior for swarms of robots, as the ability to spread throughout an environment is often the primary reason to deploy a large number of robots. There are three dispersion behaviors discussed in this work. The `disperseFromSource` behavior moves the swarm away from a distinguished source robot, which remains stationary. The `disperseFromLeaves` behavior moves the entire swarm, including the source, away from leaf robots. In this behavior, the leaf robots remain stationary. The `disperseUniformly` behavior moves all robots away from each other in order to produce and maintain uniform dispersion. It uses environmental boundary conditions to limit the final dispersion.

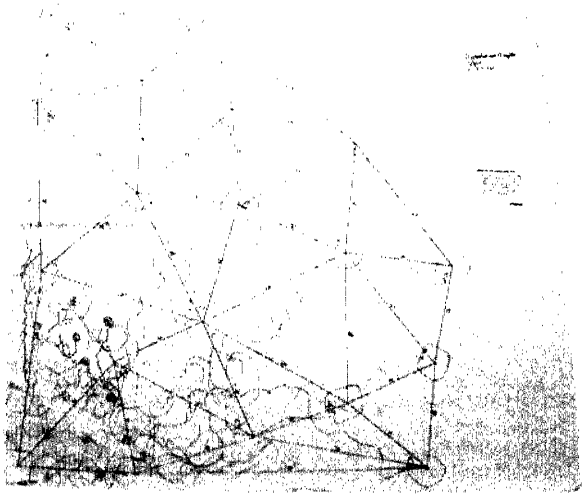
The goal of the `disperseFromSource` behavior is to expand the swarm radially from a central location. This is useful for filling a large area quickly, or deploying from a landing point.

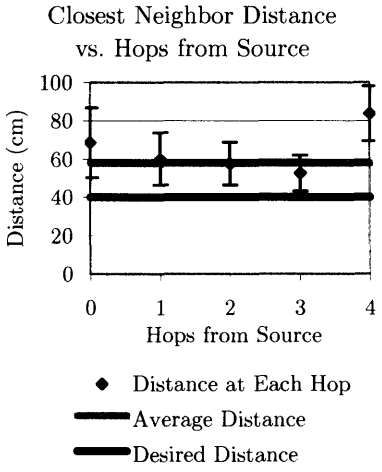
Spec:

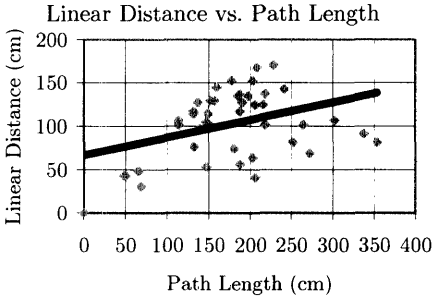
- The source robot should not move.
- All robots should be no closer than **d** to any other robot
- All non-source robots should move in a straight path at constant velocity to final dispersed position.

Figure 39: These video clips show the **disperseFromSource** behavior in action.

```
disperseFromSource(beh, gType, d)
1.  nbrSet ⇐ nbrOp(nbr.M[gType].hops ≤ self.M[gType].hops)
2.  if nbrSet ≠ ∅
3.    avoidManyRobots(beh, nbrSet, d)
4.  endif
```







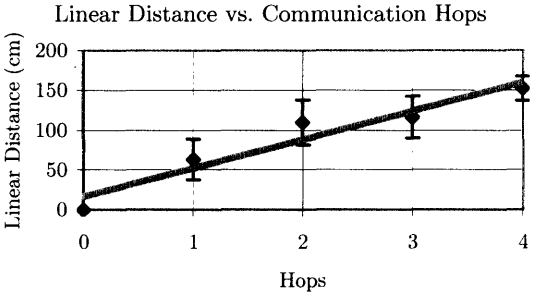


Figure 38: Data from multiple runs of **disperseFromSource**. **Top Left:** Traces from one run. The final positions have been triangulated to compute closest neighbors. **Top Right:** The inter-robot separation distances plotted vs. hops from source. The desired separation distance is 40cm, the average separation is 57 cm, and the minimum is 40 cm. Of note is the large separations of the 4-hop robots that are on the edge of the network. This is not present in **disperseFromLeaves** **Bottom Left:** Linear distance does not correlate well with path length. This is most likely due to the interference near the source. **Bottom Right:** Communication hops does scale with linear distance, due to the line-of-sight IR communications system.

Line 1 makes a set of all the robots that are at fewer or equal hops from the source than the active robot. If this set is non-empty then line 3 moves the active robot away from the average bearing to all of these robots. Figure 37 illustrates this process, and shows one active robot dispersing from its neighbors. Usually, all robots except for the source are also dispersing, and their motions sometimes interfere with each other, especially if robots in the interior cannot satisfy the range constraints. This interference will eventually push the robots on the perimeter outward, creating more space for the interior robots.

The algorithm terminates when the `avoidManyRobots` behavior is inactive, which implies that all the robots have met their distance constraints: each robot is further than **d** from its parent or sibling neighbors. This means that 1-hop robots are further than **d** from the source or any other 1-hop robot. By induction, all robots are further than **d** from all their neighbors. It is very unlikely for gradient propagation to have persistent hop errors in messages, which means that it is very unlikely that some robot always moves in the incorrect direction.

This behavior does not close "holes" in a fully dispersed swarm, Once robots have satisfied their range constraint, a void left by a removed robot will cause the surrounding robots to acquire new neighbors, all of which must be further away than the robot that was removed. The `disperseUniformly` behavior is the only dispersion behavior presented here that can close voids.

Experimental Results

The resulting group behavior is effective: all the robots move away from the source in a semi-orderly fashion, and then achieve the desired inter-robot separation distance of **d** from all downstream and lateral neighbors. Over all runs, the minimum inter-robot separation distance was 40cm. The average was 57 cm, which is to be expected because there is no constraint on the maximum separation distance. Essentially, any jostling, or sensor error will tend to only move the robots further away from each other. The average path efficiency is 64%, and there is much interference around the source as robots jostle to get into free space.

4.5.3 disperseFromLeaves

The goal of the `disperseFromLeaves` behavior is to expand the swarm, including the source robot(s), away from the leaves of the gradient tree. This is useful for exploring interior locations, including areas with constrictions and multiple paths. It can also fill a large area quickly. An external behavior is responsible for selecting the source robots.

Spec:

- The leaf robots should only move in response to other leaf robots.
- All robots should be no closer than d to any other robot.
- Move in a straight path at constant velocity to final dispersed position.

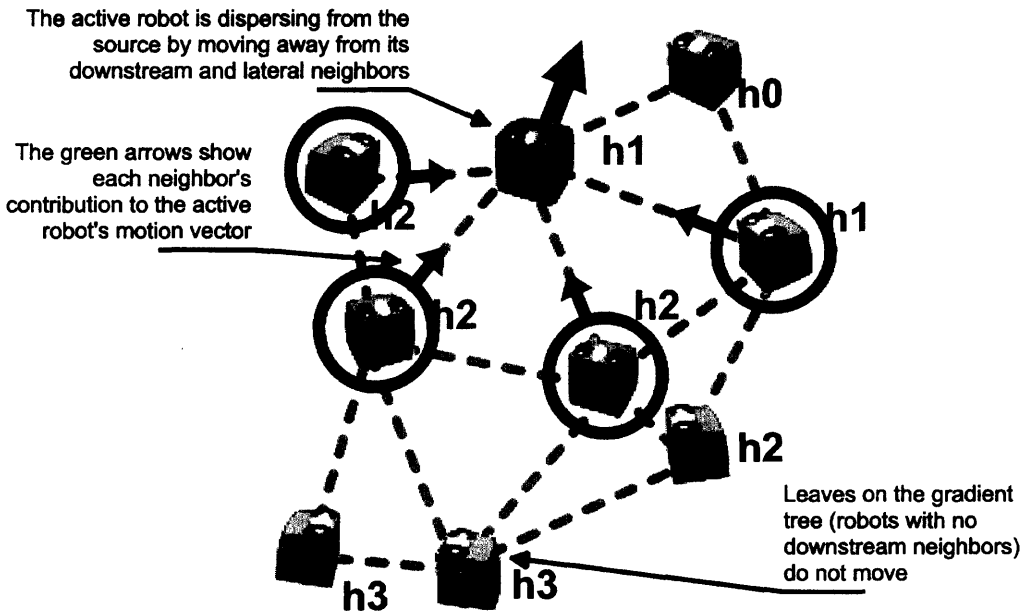


Figure 40: These robots are dispersing from the leaves of a gradient communication tree using the `disperseFromLeaves` behavior. The active robot moves away from all neighbors that are children or siblings in the gradient tree. This moves the active robot away from the leaves of the tree and towards the source.

```
disperseFromLeaves(beh, gType, d)
5.  nbrSet ⇐ nbrOp(nbr.M[gType].hops ≥ self.M[gType].hops)
6.  if nbrSet ≠ ∅
7.    avoidManyRobots(beh, nbrSet, d)
8.  endif
```

Line 1 makes a set of all the robots that are greater or equal hops from the source. If this set is non-empty then line 3 moves the active robot away from the average bearing to all of these robots. Figure 40 illustrates this process, and shows one active robot dispersing from its neighbors. Usually, all robots except for the source are also dispersing, and their motions sometimes interfere with each other, especially if robots in the interior cannot satisfy their range constraints. This interference will eventually push the robots on the perimeter outward, creating more space for the interior robots.

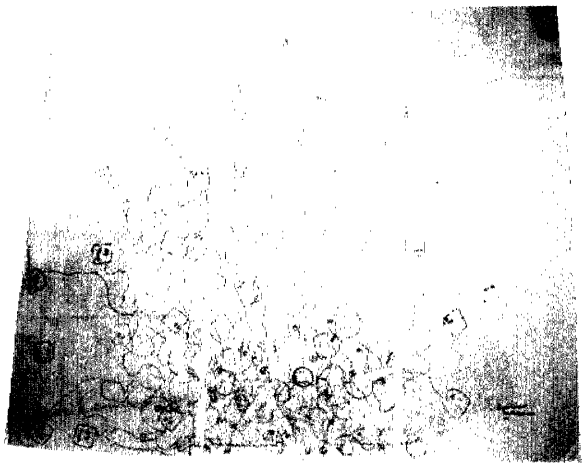
The discussion parallels that of `disperseFromSource`. The algorithm terminates when the `avoidManyRobots` behavior is inactive, which implies that all the robots have met their distance constraints, and every robot is further then **d** from all of its neighbors. Like `disperseFromSource`, this behavior does not close "holes" in a fully dispersed swarm.

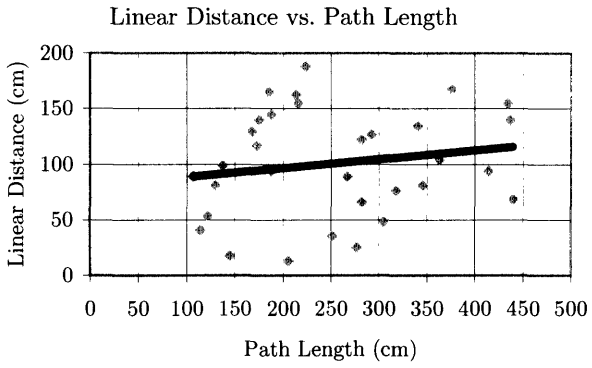
Care must be taken in source robot selection. If source robots are located in the interior of the network, then as robots move away from the leaves, they will not have anywhere to go. The worst case is when the source robot is at the center of the graph and all the leaves are an equal number of hops away, there will be very little motion at

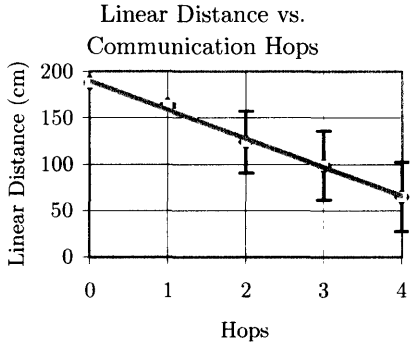
all. The directedDispersion demo in section 5.5 uses edge robots (see detectEdges in section 4.5.10) as sources.

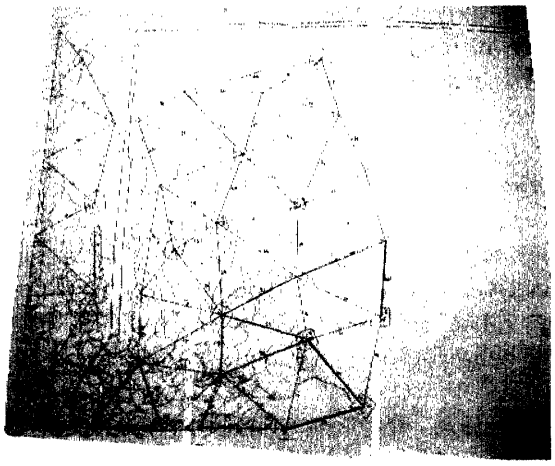
Experimental Results

The resulting group behavior is effective as long as the correct robot is selected as









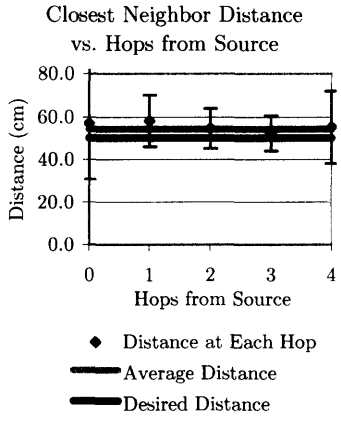


Figure 41: These traces show some results of the disperseFromLeaves behavior. After dispersing, the average inter-robot separation was within 92% of the specified separation distance. The path efficiency is 45%. **Bottom Right:** Note that the dispersion distances are much more uniform than those of disperseFromSource.

the source. Figure 41 shows data collected from experimental runs where the source was a robot that is on the edge of the swarm.

Leaf robots can move in response to each other. This is required to satisfy the dispersion distance constraint among leaf robots along the perimeter of the swarm, but weakens the useful property of leaf robots staying where they are placed – sort of like maximum extent markers. However, if leaf robots did not respond to each other, it would be possible to have an arbitrary number of them piled into a corner, which is a worse problem.

If there are multiple sources in the swarm, the Voronoi tessellation will allocate robots to sources in such a way as to encourage a breadth-first search of the entire workspace. The image in the bottom-left of Figure 41 illustrates this behavior. The robots on the left side of the plot are in a corridor, and are not constrained to disperse at the same rate as the robots in the open area to the right. However, because as each source moves forward, it moves further from its leaves, and gives the other source the opportunity to have robots move in its direction. This tends to balance the dispersion from the intersection of the two gradients, which is in the lower left of the image. This property is exploited in the Directed Dispersion application in section 5.5.

The path efficiency is 45%, which is comparable to `disperseFromSource`, and there is also a tight correlation between linear distance and communication hops. However, the correlation between linear distance and path length is worse, which can be seen in the hectic nature of the traces. The accuracy of the inter-robot separation distance from the desired separation is 92%, which is much higher than that of `disperseFromSource`. One reason is that `disperseFromSource` "pushes" robots away from the source and produces radially divergent paths. This tends to increase separation distances between sibling robots as they get further from the source, as can be seen in the upper right graph of Figure 38. The `disperseFromLeaves` behavior does not have this property.

4.5.4 diperseUniformly

The `diperseUniformly` behavior spreads the robots uniformly throughout the environment. Walls and maximum ISIS communication range are used as boundary conditions to limit the final dispersion.

Spec:

- Fill the containing workspace with robots such that the variance of the inter-robot separation distances is as small as possible.

```
diperseUniformly(beh)
1. nbrSet ⇐ nbrOp-closestN(nbrs, NUMCLOSEST)
2. beh.translationalVelocity = vTrans(nbrSet)
3. beh.rotationalVelocity = vRot(nbrSet)
```

Where `vTrans()` and `vRot()` are given by the velocity equations defined below.

$$v_{trans} = -\frac{v_{max}}{c \cdot r_{safe}} \sum_{i=1}^{c} \sin(\text{nbrList}_i.\text{bearing})(r_{safe} - \text{nbrList}_i.\text{range})$$

$$\mathbf{v_{rot}} = -\frac{\mathbf{v_{max}}}{\mathbf{c} \cdot \mathbf{r_{safe}}} \sum_{i=1}^{c} \cos(\mathbf{nbrList_i.bearing})(\mathbf{r_{safe}} - \mathbf{nbrList_i.range})$$

where:

nbrList$_i$.bearing and **nbrList$_i$.range** are the bearing and range to ith neighbor in the **nbrList** array.

$\mathbf{r_{safe}}$ is the maximum distance two robots can be separated by and still receive 80% of the ISIS packets sent between them.

$\mathbf{v_{max}}$, which is set to **beh.speed**, is the maximum speed output by this behavior.

c is equal to NUMCLOSEST, which is the number of closest neighbors to consider when computing the dispersion vector. This is an approximation to finding Voronoi neighbors and is described in detail below.

Each term in the summation is the contribution to the active robot's velocity vector from a single neighbor. This term is large if the neighbor is nearby, small if the neighbor is distant, and zero if the neighbor is further away than $\mathbf{r_{safe}}$. Figure 42 shows a plot of the magnitude of the velocity contribution from one neighbor. The direction of the velocity is away from each neighbor. The summation adds the contributions from each neighbor, then scales the computation by the number of neighbors to produce a final velocity vector.

The **disperseUniformly** behavior is essentially a relaxation algorithm; imagine compressed springs placed between neighboring robots. This will tend to expand the swarm to fill the available space, but once the space is occupied, robots will position themselves to minimize the energy in the springs. Total group energy is minimized by minimizing local contributions, which happens when all the inter-robot distances are roughly equal. A thorough treatment of this technique is presented in [3]. Physical walls and a maximum dispersion distance of $\mathbf{r_{safe}}$ are used as boundary conditions to help prevent the swarm from spreading too thin and fracturing into multiple disconnected components. Figure 46 and Figure 44c show the robots uniformly dispersed in variously sized spaces.

The closest neighbors in a graph can be found by triangulation, and are also called Voronoi neighbors, as they are neighbors of the adjoining Voronoi polygons of **robot$_i$** in the network graph **G**. Determining the set of Voronoi neighbors **vnbrs** from the set of all neighbors, **neighbors**, in real-time, is computation-intensive, [4] so an approximation is used. The closest neighbor to the robot will always be in the set

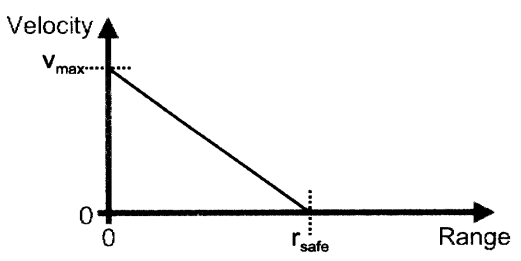


Figure 42: The **diperseUniformly** velocity equations command a velocity of v_{max} when the active robot is close to a neighboring reference robot. The commanded velocity falls to zero when the active robot is further than r_{safe} from the reference robot..

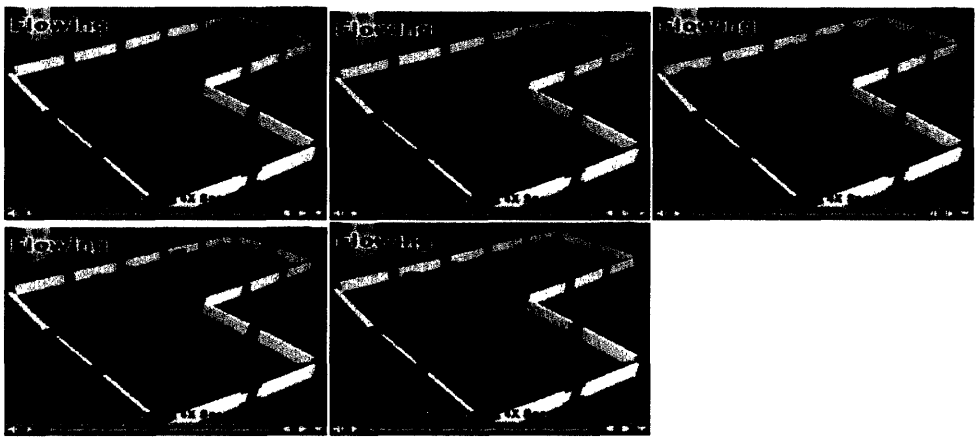


Figure 43: These video clips show the **disperseUniformly** behavior in action. For the keen of eye, robots that are near walls are flashing their blue light; robots in the interior are flashing their green light.

vnbrs. However, avoiding a single robot results in hectic movement as small sensor errors can cause the closest neighbor to change often. Increasing the number of neighbors in the approximation can cancel some of the sensor noise, but can also include many non-Voronoi neighbors and cause the dispersion errors shown in Figure 44a-h. If all neighbors are used in the set, all the robots are forced against the walls.

The **disperseUniformly** behavior can close voids in the network, but only if the environment is smaller than the maximum safe dispersion, which occurs when all robots are at a range of $\mathbf{r_{safe}}$ from their neighbors.

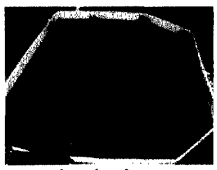
Experimental Results

In practice, using the two closest neighbors worked the best. There are some cases in which second-closest neighbor is not a Voronoi neighbor, caused when the farther neighbor is "shadowed" by the closer neighbor. This case causes the robot to move in the same direction it would if only avoiding one neighbor, which does not cause errors, but does increase jitter. This "shadowing" effect is usually short lived, as the robot will typically encounter another neighbor or obstacle quickly.

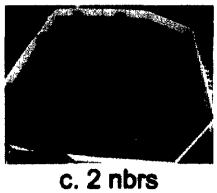
The behavior takes a long time to move the robots throughout their environment, with an average path efficiency of no more than 6%. All the robots are in constant motion, making their paths difficult to measure. An estimate based on a max velocity of 22 cm/s, and a longest path through the environment of 668 cm, and an average running time of 10 minutes yielded this result. Attempts to accelerate the motion in general cause excessive motion from interior robots. The motion is from neighbor position noise which comes from both sensor errors, packet collisions, and the two-neighbor approximation. Future work will be to compute more of the Voronoi neighbors and compare performance.

In order to compute the correctness, the ideal inter-robot separation, $\mathbf{e_{opt}}$, must be computed. This can be computed by taking the total workspace area and computing the maximum area that can be occupied by packed circles [9]. The diameter of the circles is equal to $\mathbf{e_{opt}}$:

a. 0 nbrs



b. 1 nbrs



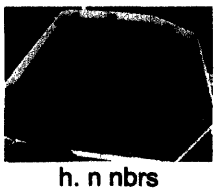
c. 2 nbrs

d. 3 nbrs

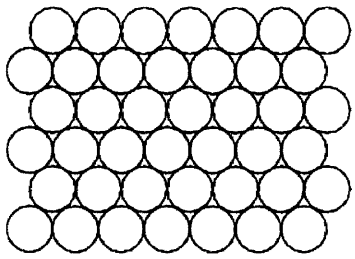
e. 4 nbrs

f. 5 nbrs

g. 6 nbrs



h. n nbrs



hexagonal packing

$$a_{packing} = \eta_h a$$

Where **a** is the workspace area and η_h is $\frac{\pi}{2\sqrt{3}} = 0.906899682...$

Robots on the perimeter of the workspace will occupy semicircles instead of circles, and this complicates using the packing efficiency to simplify the problem. Another approach is to increase the size of the workspace area as a function of the perimeter to allow full circles to be used for robots on the edge. In order to do this exactly, the final positions of each robot and the shape of the perimeter must be known. We can approximate the increase of a_{edge} by adding a series of half-circles of radius $e_{opt}/2$ around the perimeter of the workspace. (Imagine a string of pearls with each pearl cut in half)

$$a_{hc} = \frac{\pi(e_{opt}/2)^2}{2}$$

$$n_{hc} = \frac{p}{e_{opt}}$$

$$a_{edge} = a_{hc} n_{hc} = \frac{\pi p e_{opt}}{8}$$

where **p** is the perimeter of the workspace.
The area "occupied" by each robot is:

$$a_{robot} = \frac{a_{packing} + a_{edge}}{\frac{(n_{start} + n_{finish})}{2}}$$

where n_{start} and n_{finish} are the number of robots at the beginning and end of each trial. This is not constant as robots must be

Figure 44: The `diperseUniformly` algorithm is designed to spread the robots evenly. Instead of computing the closest neighbors (the neighbors of adjoining Voronoi polygons) to determine which robots to avoid it avoids the n closest neighbors, sorted by range. Figs. a-h show the results of avoiding an increasing number of neighbors, with h showing the limit. Avoiding the two closest neighbors worked best in practice.

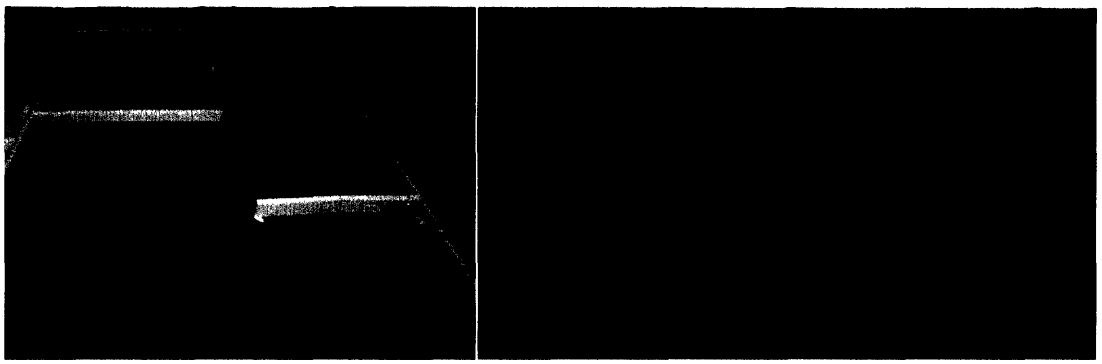


Figure 46: The picture on the left is a dispersion into the small test space used for experiments. The picture on the right shows robots dispersed in a very large room, note the person in the upper-left corner.

removed to recharge. We will replace this with $\overline{n}$.

$$a_{robot} = \frac{\eta_h a + \frac{\pi p e_{opt}}{8}}{\overline{n}} = \frac{\eta_h a}{\overline{n}} + \frac{\pi p e_{opt}}{\overline{n} 8}$$

and from the area of the packing circles:

$$a_{robot} = \pi e_{opt}^{2}$$

The optimal edge length can then be computed:

$$e_{opt}^{2} - \frac{p}{8n} e_{opt} - \frac{\eta_h a}{\pi n} = 0$$

The net result is a quadratic for e_{opt} that can be solved in closed form or by numerical methods. When $\mathbf{r_{opt}}$ is compared to the experimental data in Figure 45, the average inter-robot separation distance is 90% of the best theoretical spacing.

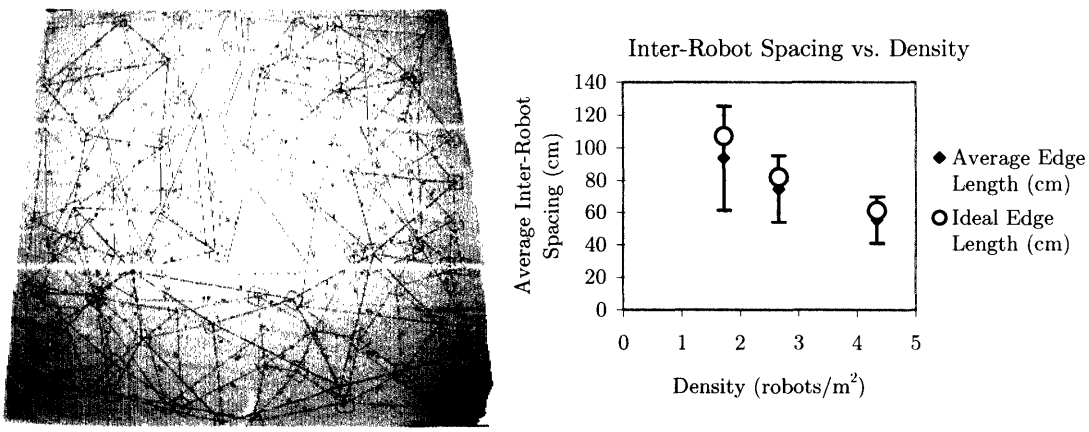


Figure 45: These traces show some results of the **diperseUniformly** behavior. After dispersing, the average inter-robot separation was 90% of the theoretical optimum. The path efficiency of this algorithm is low, as the robots are in constant motion, and was no higher than 6%.

4.5.5 followTheLeader(-data)

The `followTheLeader` behavior dynamically constructs an ordered line of robots. This line is suitable for leading a group of robots into an area. Another behavior is required to control the leader.

Spec

- Form a graph of n robots using n-1 edges. (A line of robots)
- Maintain a specified maximum edge length for each edge in this graph.

Figure 47 shows a diagram of this behavior. The pseudocode for this algorithm is somewhat long, and is shown on the next page. The application program passes in the inputs described below:

`beh`	The behavior output struct.
`lengthInput`	The total length of the line. Only used by the line leader.
`d`	The inter-robot dispersion distance.
`lineLeader`	A Boolean. If true, sets this robot as the line leader.

Table 4: Follow the leader input parameters.

In addition, there are neighbor data variables that are required to form the line:

`<leaderID>`	The robotID of the leader of the current robot
`<followerID>`	The robotID of the follower of the current robot
`<length>`	The length of the line
`<order>`	The order of the current robot in the line
`<lineLeaderID>`	The robotID of the line leader

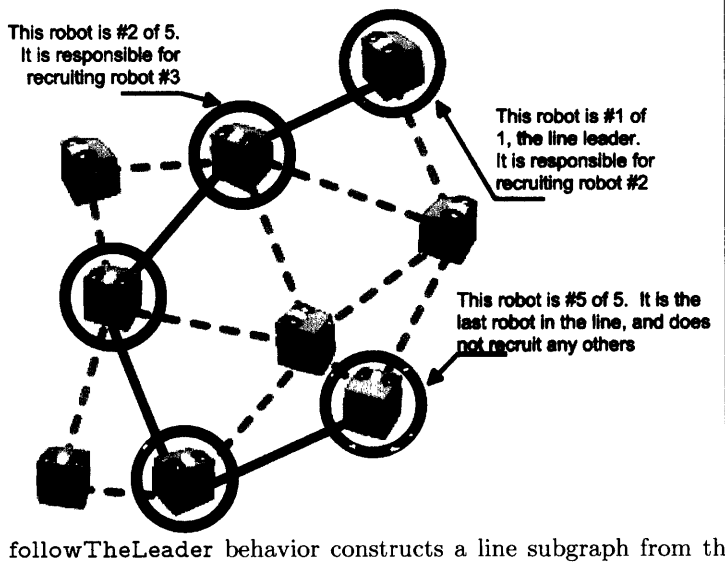


Figure 47: The `followTheLeader` behavior constructs a line subgraph from the total group of robots. There is one distinguished leader robot that is responsible for recruiting the first follower. Each successive follower recruits another until the line is the desired length.

Table 5: Follow the leader neighbor data byte variables.

followTheLeader(beh, lengthInput, d, lineLeader)

```
defineNbrVar ⟨leaderID, followerID, length, order, lineLeaderID⟩
if lineLeader = TRUE
  leaderID ⇐ ∅
  length ⇐ lengthInput
  order ⇐ 1
  lineLeaderID ⇐ MYROBOTID
else
  leaderNbr ⇐ nbrOp-ID(*, nbr.followerID = MYROBOTID)
  if leaderNbr ≠ ∅
    leaderID ⇐ leaderNbr.robotID
    length ⇐ leaderNbr.length
    order ⇐ leaderNbr.order + 1
    lineLeaderID ⇐ leaderNbr.lineLeaderID
    followRobot(beh, leaderNbr, d)
  else
    leaderNbr ⇐ nbrOp-ID(*, nbr.followerID = ANYROBOTID)
    if leaderNbr ≠ ∅
      leaderID ⇐ leaderNbr.robotID
    else
      leaderID ⇐ ∅
    endif
    length ⇐ 0
    order ⇐ 0
    lineLeaderID ⇐ ∅
  endif
endif

if length > order
  if (followerID ≠ ∅) and (followerID ≠ ANYROBOTID)
    followerNbr ⇐ nbrOp-ID(*, followerID)
    if (followerNbr ≠ ∅) and (followerNbr.leaderID = MYROBOTID)
      /* The previous follower is still there and following */
    else
      followerNbr ⇐ ∅
    endif
  endif

  if (followerNbr = ∅)
    followerNbr ⇐ nbrOp-Closest(*, nbr.leaderID = MYROBOTID)
    if followerNbr ≠ ∅
      followerID ⇐ followerNbr.robotID
    else
      followerID ⇐ ANYROBOTID
    endif
  endif
else
  followerID ⇐ ∅
endif
```

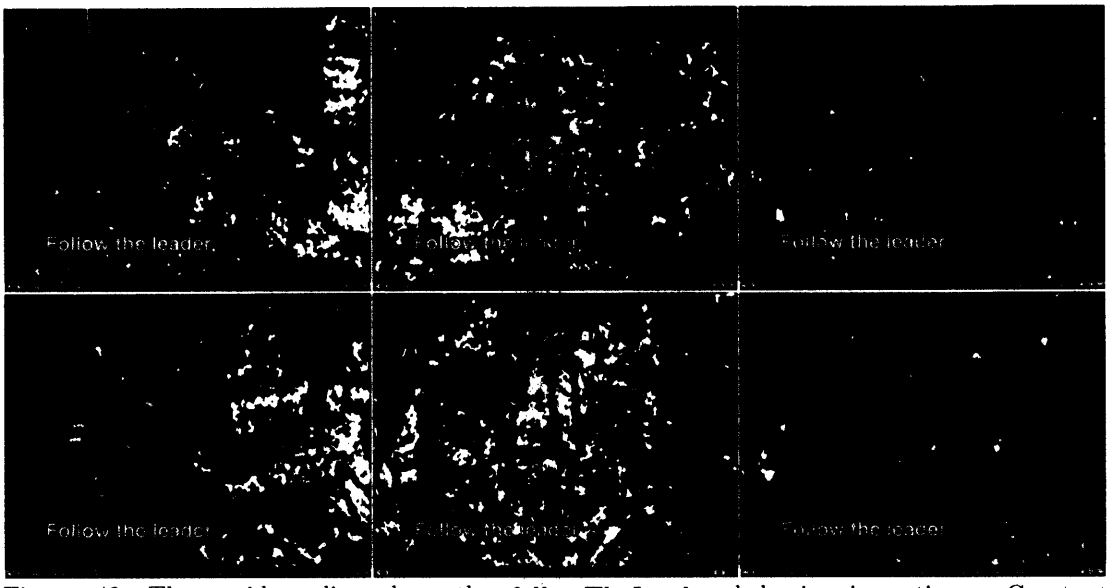


Figure 48: These video clips show the **followTheLeader** behavior in action. Constant handshaking between successive robots keeps the line robust. If a robot does not respond, another is recruited to take its place. The chain breaks in frame four, re-forms in frame five, and is stable in the final frame.

Each robot is responsible for recruiting the next robot in the line. This process continues until it gets to the last robot, which does not recruit a follower. The recruitment process is controlled by the set of neighbor variables described in Table 5. The code is divided into two sections. The top section (lines 1-26) is responsible for finding and following a leader robot, and is divided into three highlighted subsections. The second section (lines 27-46) is responsible for recruiting a follower robot.

Section 1: Following a Leader

Line 1 creates the neighbor variables described in Table 5. Lines 3-6, the first subsection (red highlight), are executed if this robot is the line leader. Line 3 invalidated the leaderID, because this robot has no leader. Line 4 copies the **lengthInput** input parameter into the **length** neighbor variable. Line 5 sets the order, and line 6 copies this robot's robotID into the **lineLeaderID** neighbor variable.

If this robot is not the line leader, then it looks for its leader, i.e. any nearby robot that has its **followerID** variable set to the active robot's robotID. Line 8 does this comparison. If one is found, lines 10-14, the second subsection (green highlight), are executed to update the active robot's state in lines 10-13 and follow the leader in line 14.

If no leader is found, the third subsection (blue shading), looks for a neighbor that is recruiting a follower. The neighbor advertises this by broadcasting ANYROBOTID in its **followerID** neighbor variable. If a neighbor is recruiting, then the active robot copies that robot's robotID into its **leaderID** neighbor variable. This will be used in line 30 in the leader robot. If there is no neighbor recruiting, then line 20 clears the leaderID neighbor variable. In either case, there is no robot to follow and this robot is not part of a line, so lines 22-24 reset all neighbor variables.

Section 2: Recruiting a Follower

The second section (lines 27-46) will recruit a follower robot if necessary. Line 27 checks to see if a follower is needed. If not, then line 45 invalidates the `followerID` neighbor variable, which stops the recruitment process.

If a follower is needed, lines 28-30 check to see if the previous follower is still visible and still reporting that the active robot is its leader. If not, line 33 invalidated the `followerNbr` neighbor struct.

If `followerNbr` is null, either because it has never been set or because it has been cleared in line 33, then this robot need to recruit a follower. Line 37 checks to see if any neighbor has responded to a previous recruitment attempt. If so, then line 39 completes the recruitment of this neighbor by setting the `followerID` neighbor variable. This will interact with line 8 in the neighboring robot, and allow it to start following. If there have been no responses to the recruitment, line 41 sets to the `followerID` to ANYROBOTID to try and recruit a follower.

Once the link is established, the leader and follower are handshaking continuously with their `followerID` and `leaderID` neighbor variables. If the leader does not see its follower, it will recruit another. This constant handshaking lets the line re-form quickly if robots lose communications or encounter physical obstacles that preventing them from following their leader.

Each recruitment takes three neighbor cycles. First the leader has to advertise for a recruiter, that robot needs to respond, then the leader needs to acknowledge that response before the follower starts following. The maximum time to propagate the recruit packets from the line leader to the last robot is

$$3 \cdot \text{lineLength} \cdot t_n$$

Experimental Results

This behavior works well, and is quite a crowd pleaser. The constant handshaking and re-recruiting makes the behavior robust in spite of communications failures and physical obstacles. A variant of the algorithm where follower robots follow any robot that is recruiting if they do not have a leader yet reduces the time to form the line to $\text{lineLength} \cdot t_n$, but can cause interference if multiple robots respond to the same leader.

4.5.6 orbitGroup

The `orbitGroup` behavior directs an active robot to move in a path around the perimeter of a group of reference robots. This is useful for perimeter surveillance or general-purpose group navigation.

Spec

- Move an active robot around a designated group of reference robots along a path offset a distance d from the perimeter of the group. This path should be tight, tracking both convex and concave curves of the group. It should be no closer than d and no further than $(\sqrt{2}/2)d$ from any reference robot.
- The active robot should move with constant velocity along this path.

```
orbitGroup(beh, d, orbitDir)⟨orbiter⟩
1.  if orbiter = TRUE
2.     nbr ⇐ nbrOp-Closest(*, nbr.orbiter = FALSE)
3.     orbitRobot(beh, nbr, d, orbitDir)
4.  endif
```

Line 1 checks to see if the current robot is an active robot (*orbiter* = TRUE) or a reference robot. If the robot is active, line 2 finds the closest reference (non-orbiting) robot. Line 3 uses **orbitRobot** to orbit this reference robot in the specified direction. While the active robot moves around the network, the reference robot changes as the active robot becomes closer to other neighbors. This results in the "puffy cloud path" around the perimeter of the network shown in Figure 49. This path is similar to what would be created by using the right hand rule to escape from a maze. This path is different from the convex hull of the reference group, and will be the same only if there are no concavities. Even if the reference group has concavities, the **orbitGroup** path is bounded from outside by the convex hull.

Limitations

If an orbiter is started from the interior of the group, it might not ever be able to move to the outside edge to start the group orbit. As it moves, its closest reference robot might always be an interior robot, and the active robot would get stuck in an internal limit cycle and never reach the perimeter. Some bootstrapping to guide the active robot to the perimeter of the network by using a combination of **edgeDetection** from section 4.5.10 and **navigateGradient** from section 4.5.7 could eliminate this problem.

Because the active robot travels in a circular path around each pivot robot, the path generated by the **orbitGroup** behavior is longer than the optimal path, but within a multiplicative constant of $\pi/2$. Figure 50 shows the **orbitGroup** path and optimal path for reference robots arranged in a straight line, a convex curve, and a concave

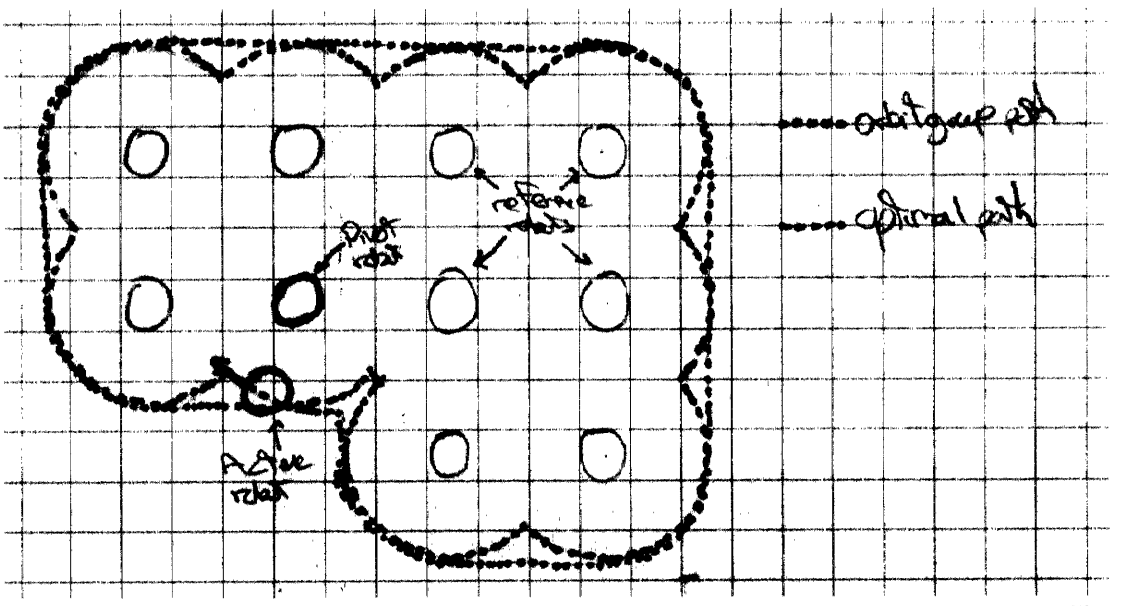


Figure 49: The **orbitGroup** behavior produces the "puffy cloud path" shown in blue above. The red path is the optimal path around the perimeter of the network. The ratio between the two is bounded by a constant factor.

curve.

Case 1: Reference robots in straight line:

$$s_{opt} = d$$

$$s_{pcp} = 2r \arcsin\left(\frac{d}{2r}\right) \leq \pi r$$

$$\text{when } d = 2r$$

$$\eta = \frac{2r}{\pi r} = \frac{2}{\pi}$$

Case 2: Reference robots in convex curve of angle θ:

$$s_{opt} = r\theta$$

$$s_{pcp} = r\theta$$

$$\eta = 1$$

Case 3: Reference robots in concave curve of angle ϕ:

$$s_{opt} = 2d - 2\sin\left(\frac{\phi}{2}\right)$$

$$s_{pcp} = r\left(4\arcsin\left(\frac{d}{2r}\right) - \phi\right)$$

$$\text{max when } \phi = 0 \text{ and } d = 2r$$

$$s_{pcp} \leq 2\pi r$$

$$s_{opt} = 2d = 4r$$

$$\eta = \frac{2}{\pi}$$

In all cases, the path from the `orbitGroup` behavior has a theoretical minimum efficiency of $\frac{2}{\pi}$, or 63%.

Experimental Results

Figure 50 shows traces of three different orbits around a group of reference robots, and the associated path data for each run. The measured path efficiency is 87%, with an accuracy of 71%. The main source of error in the paths is when the active robot needs to switch references. The neighbor cycle period is 250 ms. At an average speed of 21 cm/s, that is about 5 cm of lag before new position information is available from the reference robots. The puffy cloud path already brings the active robot close to the

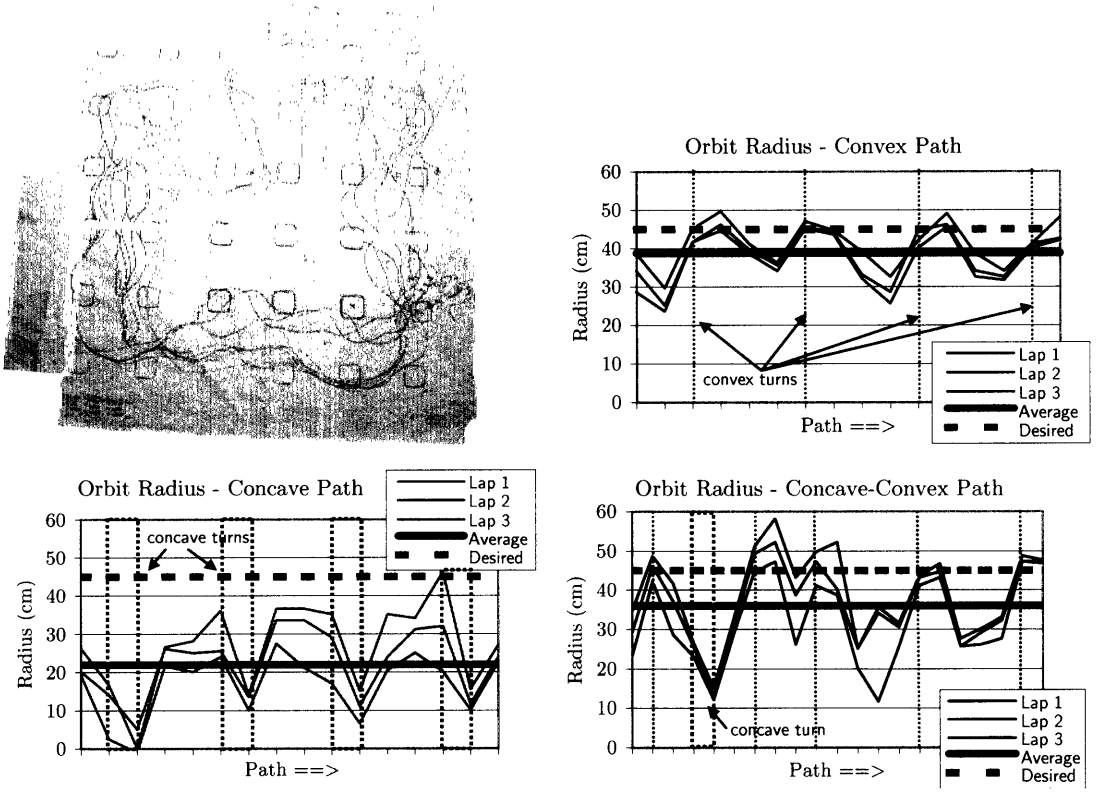


Figure 50: Traces from three different paths using the `orbitGroup` behavior. The measured path efficiency is 87%, with an accuracy of 71% **Top Left:** The traces. **Top Right:** The green run has all convex turns. The robot tends to increase its orbiting radius during these turns. **Bottom Left:** The blue run has all concave turns. The active robot does not switch references fast enough to maintain he desired radius, and even has some collisions with reference robots. **Bottom Right:** The red graph has five convex turns and one concave turn.

references, this extra distance can bring the active robot too close and cause collisions. This does not usually cause failure, because after the `bumpMove` behavior is finished, the stability of the `orbitRobot` behavior puts the active robot back on the correct path.

One possible improvement would be to sum the vector fields produced by the `orbitRobot` behavior. This would tend to smooth the paths in between robots, and add some predictive ability for concave corners.

4.5.7 navigateGradient

The goal of this behavior is to provide a general-purpose navigation algorithm capable of directing any robot to any other robot in the swarm. This "physical routing protocol" is the foundation for many other behaviors. The approach is to use stationary reference robots as navigational cues to direct the active robot towards the source of the communications gradient. In some respects it is similar to clustering, except that only the active robot moves toward the source.

Spec:

- Move to a position that is no greater than **d** from the source robot.

- Move in a straight path at constant velocity to the final position

navigateGradient(beh, gType)

```
1.  nbrSet ⇐ nbrOp(*, nbr.M[gType].hops < self.M[gType].hops)
2.  nbrSetAvoid ⇐ nbrOp(*, nbr.M[gType].hops > 0)
3.  if nbrSet ≠ {∅}
4.     beh1 ⇐ EMPTYBEH
5.     beh2 ⇐ EMPTYBEH
6.     beh3 ⇐ EMPTYBEH
7.     b ⇐ computeAverageBearing(nbrSet)
8.     rotateToAngle(beh1, b)
9.     moveForward(beh2)
10.    interstitialAvoid(beh3, nbrSetAvoid)
11.    beh ⇐ sumBehaviors(beh1, beh2, beh3)
12. endif
```

This behavior assumes that a communication gradient of type **gType** is present. The active robot runs the code above and navigates towards the source. All other robots are references.

Line 1 creates a set of neighbors that are parents of the active robot on the gradient tree. Line 2 creates a set of all neighbors that are not the source. These are robots that should be avoided by the active robot as it makes its way to the source. Line 3 checks to see if any parent robots exist. If so, line 7 computes the average bearing to this set of neighbors, and line 8 rotates the active robot in that direction. This is shown in left hand side of Figure 51. Line 10 uses a behavior called `interstitialAvoid` that tries to move the active robot through gaps in the swarm in an attempt to reduce the number of collisions. This behavior is not described in this work. The outputs of these two behaviors plus `moveForward` from line 9 are summed in line

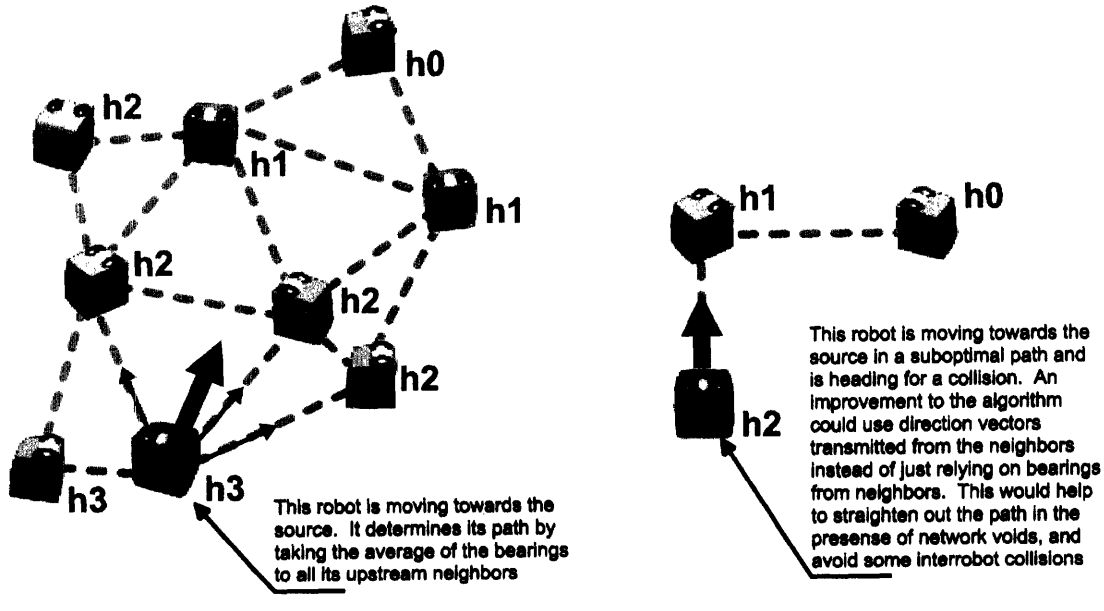


Figure 51: The `navigateGradient` behavior guides the active robot to the source of a gradient.

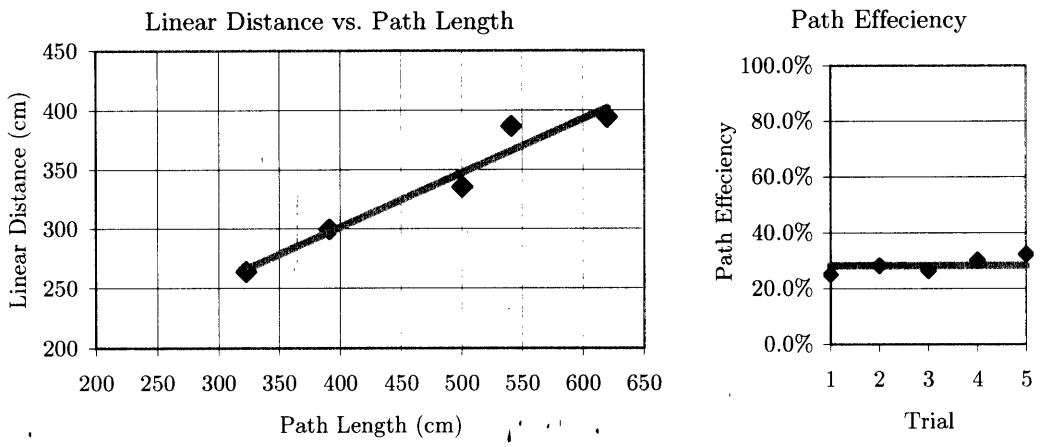


Figure 52: The **navigateradient** behavior shown a tight correlation between linear distance and path length. Average path efficiency is 28%

11.

The characteristics of gradient communication guarantee that every robot that receives a gradient message and is not a source will be in contact with at least one parent neighbor. The line-of-sight local communication constraint of the ISIS system ensures that an upstream robot will be physically closer to the source of the gradient. Therefore, any active robot can construct a path to an upstream position, and by induction, to the source. Any walls and obstacles that block communications will cause the gradient messages to route around them in order to reach the active robot. This will guide the active robot around these same obstacles on its way to the source. This can be beneficial when the obstacles are real, but communications problems or voids in the network will evoke the same response.

If the robot moves along this path at a constant velocity, it will reach the source in time bounded below by

$$\mathbf{t} = \frac{\mathbf{s}}{\mathbf{v}}$$

where **s** is the best straight line path and **v** is the velocity of the robot.

Unfortunately, this lower time is difficult to achieve in practice. The actual time is strongly affected by physical interference from neighboring robots and voids in the network. The arrangement shown in the right hand side of Figure 51 displays both of these problems. The optimal path is diagonally up and to the right, but the average of the bearings of the active robot's neighbors is straight up, towards the h1 reference

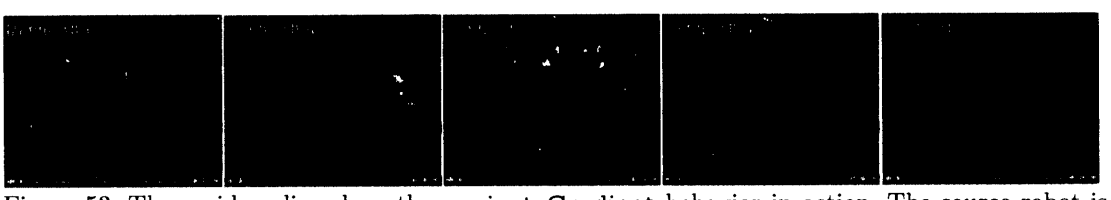


Figure 53: These video clips show the **navigateGradient** behavior in action. The source robot is wearing a small flag and is located in the bottom center of the images. The active robot is highlighted in green.

Figure 54: These traces show the results of the **navigateGradient** behavior from different starting locations. This behavior produced an average path efficiency of 28%.

robot. In the worst-case scenario where the h1 reference robot is at the limit of its communication range with the h0 robot, the active robot will not receive communications from the h0 robot until it is in approximately the same location as the h1 robot. This will cause a collision between the active robot and the h1 robot, further diverting it from the optimal path.

Experimental Results

In the video captures above, the robot highlighted in green is the source of the communications gradient, and the robot highlighted in red is moving towards it. This behavior is almost always successful in guiding the active robot to the reference. However, the average path efficiency for the trials was 28%, with errors seemlying split between collisions and network voids. A possible improvement to the navigation algorithm could have reference robots transmit direction vectors to sources. This would help to straighten out the active robot's path in the presence of network voids, and could help avert some collisions as well.

4.5.8 clusterOnSource

The goal of the clustering behavior is to move the swarm to a centralized location in as small an area as possible. This behavior could be useful for moving large objects, focusing sensors on a single stimulus, or simply collecting all the robots in one spot to put them to bed for the evening.

Figure 55: These video clips show the **clusterOnSource** behavior in action.

Specs:

- Form a single cluster of minimum size - all robots should be within a radius r from the source, where $r_{opt} < r < c_c r_{opt}$
- Robots should move along minimum shortest path to final position at constant velocity

```
clusterOnSource(beh, gType, d)
1.  nbr ⇐ nbrOp-Closest(*, nbr.M[gType].hops < self.M[gType].hops)
2.  if nbr ≠ ∅
3.    followRobot(nbr, d)
4.  endif
```

Line 1 forms a set of parent neighbors in the gradient tree, then selects the closest one. Line 3 moves the active robot towards the closest parent neighbor. This implementation is simple, but effective. The active robot in Figure 56 has three upstream neighbors, circled in green. It selects the closest of these, and then moves towards it until it is within range d. By then, it will probably be able to communicate with another robot that is closer to the source, and the process will repeat. The net result is that the entire group of robots converges towards the source. Since the range of the neighbors and the gradient hop counts are continuously updating, the behavior remains effective in the face of the radically changing network topology that the behavior causes.

Nothing in the implementation guarantees that robots will maintain connectivity as they are clustering. It is common for robots to have communications errors or mobility obstructions that prevent them from keeping up with their parent neighbors. When this happens, the swarm splits into two separate groups and does not converge onto the same source robot. The addition of "flow control" to this behavior to halt upstream robots if downstream neighbors fall behind eliminates this problem, and is employed in the **clusterWithBreadCrumbs** behavior in the next section.

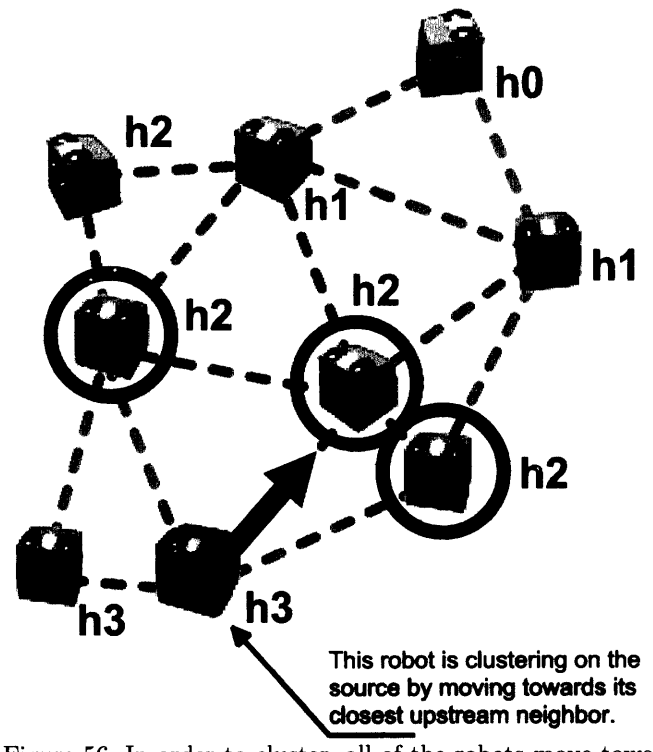


Figure 56: In order to cluster, all of the robots move towards their closest upstream neighbor. The arrow pointing away from the active robot indicates its path towards its reference neighbor. The nearest neighbor is updated every communications cycle.

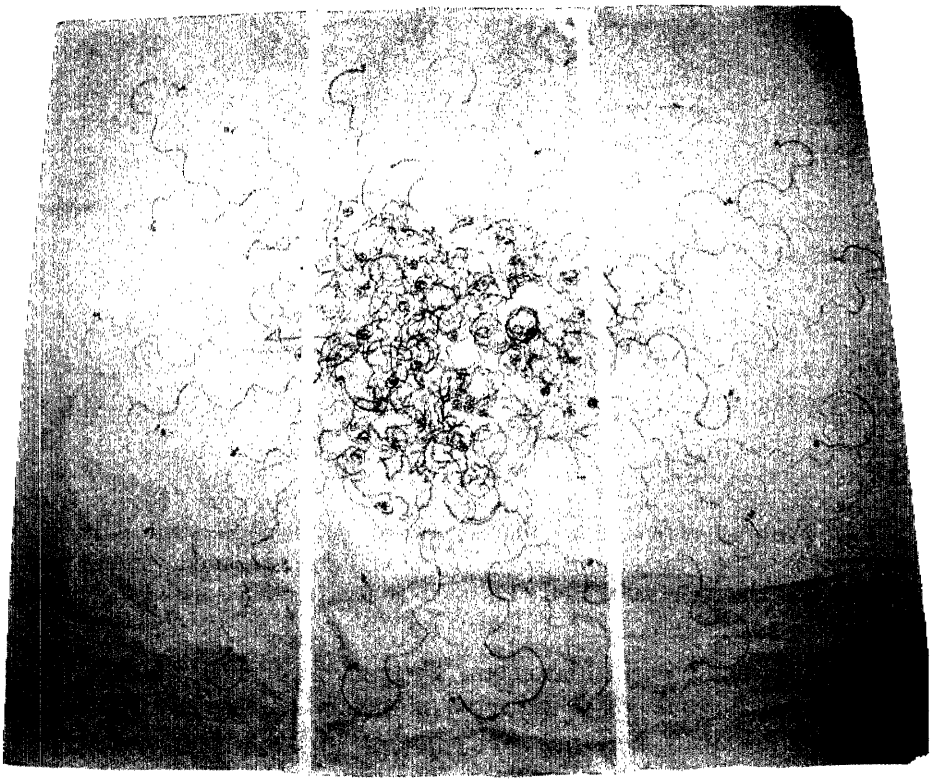


Figure 57: These traces show results of the `clusterOnSource` behavior. The outline of the source robot is located in the center of the densely marked section.

Experimental Results

Figure 57 shows the traces left by the robots during an execution of the `clusterOnSource` behavior. Figure 58 shows data on the path efficiency. Figure 59 shows the final clustering, and the data collected on packing efficiency. To measure packing efficiency, the convex hull of the final cluster was traced and compared to the

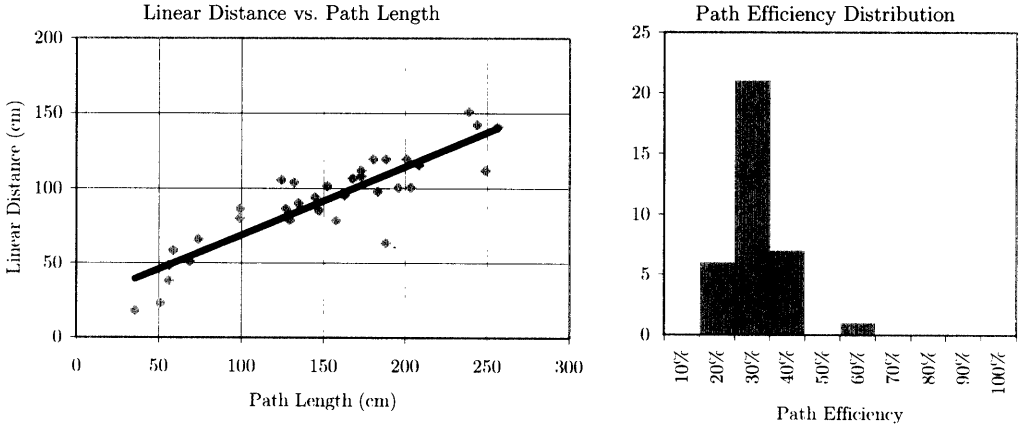


Figure 58: The path length traveled during clustering correlates well with the linear distance between the start and end points for each robot, although the average path efficiency was only 26%.

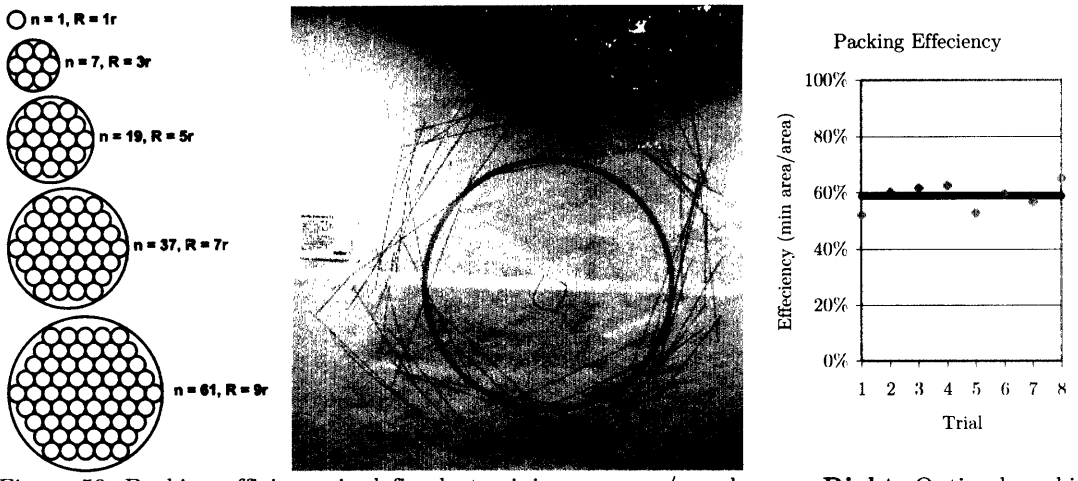


Figure 59: Packing efficiency is defined at minimum area / used area. **Right:** Optimal packing efficiency is achieve when robots form a tightly packed hexagonal pattern. **Middle:** The convex hulls formed by the clustering algorithm are shown as different colored outlines, and an optimal circle of radius R is shown in gray. **Left:** The average packing efficiency of the `clusterOnSource` behavior was 59%.

ideal hull of an optimally packed hexagon. The packing efficiency is 59%, which is fairly high, given the random nature of the robot motion when they are that closely packed. In general, this behavior performs well, but often splits the swarm into multiple disconnected components, even when clustering over short distances.

4.5.9 clusterIntoGroups

The `clusterIntoGroups` behavior implements a primitive form of division of labor. It operates in two `steps; first, each robot selects a group to join, then the behavior moves robots in the same groups together, while moving entire groups away from each other. It is used in the Swarm Choir demo in section 5.4 to separate robots based on the instrument they are playing.

Spec

- Form i groups, with each group containing exactly $\lfloor n/i \rfloor$ or $\lceil n/i \rceil$ robots. This grouping should be maintained even as population size changes.
- The distance between any two robots in the same group should be less than the distance between any two robots in different groups
- Robots should move along the shortest path to final position at constant velocity

clusterIntoGroups(beh)⟨groupGradientType⟩

```
1.  defineNbrVar ⟨grouped⟩
2.  gradientSource(self.M[groupGradientType], LATERALINHIBITION)

3.  if self.M[groupGradientType].sourceID = MYROBOTID
4.    grouped ⇐ TRUE
5.  else
6.    groupLeaderNbr ⇐ nbrOp-any(*, nbr.M[groupGradientType].hops = 0)
7.    if (groupLeaderNbr ≠ ∅) and (groupLeaderNbr.range < GROUPEDRANGE)
8.      grouped ⇐ TRUE
9.    else
10.     navigateGradient(beh, groupGradientType)
11.     grouped ⇐ FALSE
12.   endif
13. endif

14. if grouped = TRUE
15.   nbrSetGrouped ⇐ nbrOp(*, nbr.grouped = TRUE)
16.   nbrSetInMyGroup ⇐ nbrOp(*, nbr.groupGradientType = groupGradientType)
17.   nbrSetGroupedInOtherGroups ⇐ nbrSetGrouped − nbrSetInMyGroup
18.   nbr ⇐ nbrOp-closest(nbrSetGroupedInOtherGroups, TRUE)
19.   avoidRobot(nbr)
20. endif
```

The neighbor variable input **groupGradientType** is the index of the group gradient message that this robot will respond to. Each group has its own group gradient, so this sets the group this robot is a member of. Line 1 creates a neighbor variable **grouped**, that announces whether or not this robot is physically located near the rest of its group. Line 2 sources a gradient with the index **groupGradientType**. This gradient has lateral inhibition enabled, so it will only have one unsuppressed source in the swarm. This unsuppressed source will be the group leader.

Lines 3-5 check to see if this robot is the unsuppressed source of the group gradient. If so, then it is automatically clumped. Lines 6-8 check to see if this robot is within a constant range from the group leader. If so, then it is grouped. If not, lines 10-11 move the robot towards the group leader.

The last section of code moves groups away from each other. Line 14 checks to see if this robot is grouped. If so, lines 15-18 find the closest neighbor that is also grouped, but in a different group. Line 19 moves the robot away from this neighbor. This takes advantage of the self-stabilizing nature of the grouping motion – if the motion from the **avoidRobot** behavior accidentally moves this robot away from the group leader, then its **grouped** status will change, and it will move back towards the leader. If it moves back to exactly the same place, the process will repeat, but this is unlikely, as randomness in the environment will cause a different result each time. Eventually, the entire system will settle into a solution in which all robots have their conditions met and are stationary.

Limitations

This behavior falls short of the desired specification of groups being dynamic. There is no machinery in the psuedocode to dynamically select or change groups. Once a robot selects its group, it is in that group for the rest of the execution. This is the only swarm behavior that is not self-stabilizing in this way.

The physical motion component of the behavior does not have this drawback. Robots detect their group status and move appropriately, however, the maximum size of each group is currently limited by the physical number of robots that can group around the leader and satisfy the range comparison in line 6. In order to support groups of arbitrary size this comparison would need to allow robots more than one hop away from the leader to be considered grouped.

Experimental Results

Figure 60 shows clips from a video of the behavior in action. Figure 61 shows the traces left after a run, and the average group and inter-group spacing. In general, the algorithm works, but there is much wasted motion. The path efficiency is 14%, which is much lower than other clustering algorithms, and lower than that of the `navigateGradient` behavior that moves the robots towards their group leaders. There is a great deal of inter-robot interference as robots move to their final positions.

The final groups are dependent on the initial positions of the group leaders. A behavior to move these leaders towards the outside of the group would help with the interference problem. A greater concern is that the group population is not dynamic, and robots will not move to equalize group sizes

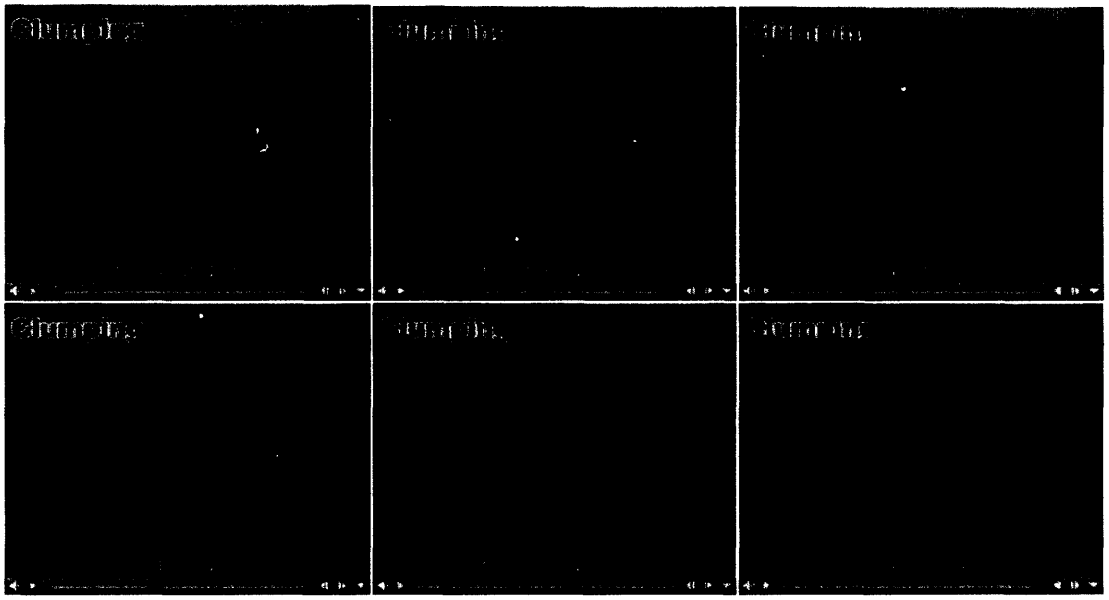
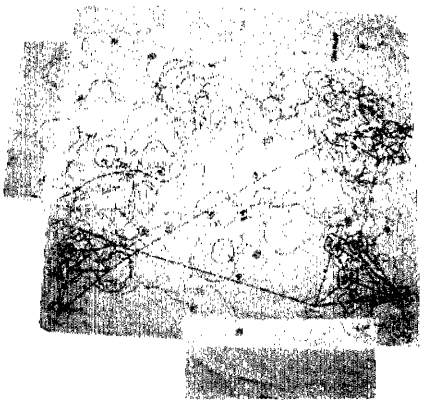


Figure 60: These video clips slow the `clusterIntoGroups` behavior in action. There are three groups, red, green, and blue. Each group elects a leader, and robots use the `navigateGradient` behavior to move towards the leader of each group. Once robots are grouped, entire groups move away from other nearby groups.



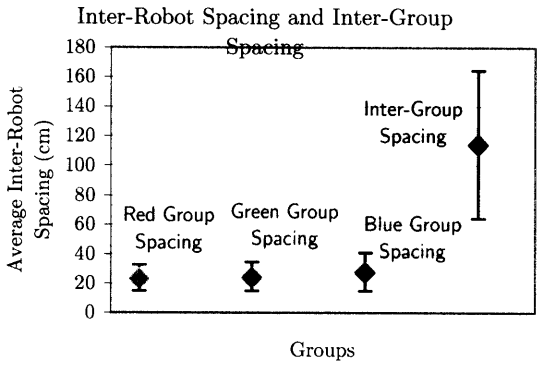


Figure 61: **Left:** These traces show the results of the `clusterIntoGroups` behavior. **Right:** The graph shows group spacing and inter-group spacing.

4.5.10 detectEdges(-polish, -data)

Determining which robots are on the edge of the network can have many useful applications. It allows you to directly compute and measure the perimeter, which can be uploaded to the user, or used for inter-swarm navigation. Tracking targets as they cross edge robots can let a surveillance application know when targets that are being tracked enter or leave the coverage area.

Spec

- Each robot determines independently whether or not it is are on the edge of the network.
- Every robot in the convex hull of the graph should be in the set of edge robots.
- Concavities larger that the average inter-robot distance should be detected as edges

edgeDetection() returns Boolean

```
edgeNbrSet ⇐ nbrOp(*, TRUE)
if ISISRadar.signal > VIRTUALNEIGHBORRADARTHRESHOLD
   edgeNbrSet ⇐ edgeNbrSet ∪ createVirtualNbr(ISISRadar.bearing)
endif

edgeNbrSet ⇐ sortNbrsByBearing(edgeNbrSet )
maxAngle ⇐ edgeNbrSet[1] + (360 - edgeNbrSet[length(edgeNbrs)] )
for i ⇐ 2 to length(edgeNbrSet) − 1
   a ⇐ edgeNbrSet[i] - edgeNbrSet[i - 1]
   if a  > maxAngle
      maxAngle ⇐ a
   endif
endfor

if maxAngle > EDGEANGLE
   return TRUE
else
   return FALSE
endif
```

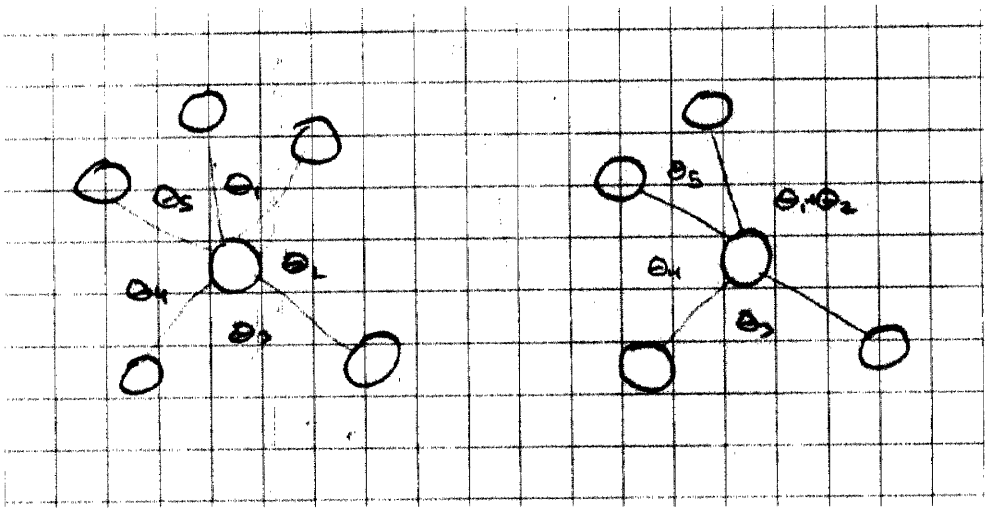


Figure 62: Robots use the bearing differences between each of their neighbors to determine weather or not they are on the edge of the network. **Left:** The separation angle θ_2 is the largest, but is insufficient to declare this robot to be on the edge. **Right:** With one neighbor removed, the largest angle becomes $\theta_1 + \theta_2$. This angle is large enough to declare the active robot to be an edge.

Line 1 puts all the neighbors into a set. Lines 2-4 check to see ifthe active robot is near a wall. If so, line 3 adds the wall to the `edgeNbrSet` as a virtual neighbor. The need for this is described below. Lines 5-12 sort the set `edgeNbrSet`, then look for the largest difference in bearing angles between any two neighbors. This "separation angle" is stored in maxAngle. Lines 13-17 compare maxAngle to a threshold and return the appropriate value.

An example of the inter-neighbor separation angle computation can be seen in Figure 62. If the largest angle is greater that a constant threshold, EDGEANGLE, then the active robot will consider itself to be on the edge of the network. However, robots near walls will have no neighbors where the walls are, which will cause them to become edges. This is appropriate for some applications, but not for others. For example the directed dispersion application in section 5.5 requires that robots near walls not declare themselves to be edges[6]. The solution is in lines 2-4, which creates a "virtual neighbor" in the direction of the closest wall. This breaks up the large empty space that the wall would otherwise create, but still allows robots who are near walls to become edge robots if they are at the front of a column of reference robots.

The parameter EDGEANGLE must be tuned by the user to achieve good results. A further complication is that the optimal calculation of an edge is highly subjective and depends strongly on the density of the robots, the desired minimum feature size, and specific environmental details. In particular, ideal concave edges are difficult to define, and require the user to experiment with the application to determine what produces the best result.

Experimental Results

In practice, a value of 220 degrees was effective at eliminating most false positives, while still providing edges for the frontiers needed in the directedDispersion algorithm in section 5.5.

[6] Jennifer Smith at iRobot is responsible for the observation of the problem and the solution.

4.6 Summary

The ISIS inter-robot communications is a good foundation for inter-robot positioning and communications. The gradient messaging system provides an sound abstraction upon which to build many different group behaviors. Behaviors use the network formed by the gradients, which usually modify the network. The interesting feedback between the network graph and the behaviors is exploited in many of these behaviors. In order to predict what the behaviors will do, invariants must be found not only across multiple executions, but also across all allowable sets of graphs.

	Path Efficiency	Goal Correctness / Efficiency
orientToRobot	-	98%
matchHeadingToRobot	-	97%
followRobot	114%	97%
avoidRobot	69%	93%
orientForOrbit	-	84%
orbitRobot	93%	93%
avoidManyRobots	68%	76%
disperseFromSource	64%	69%
disperseFromLeaves	45%	92%
disperseUniformly	6%	90%
orbitGroup	87%	71%
navigateGradient	28%	100%
clusterOnSource	26%	59%
clusterIntoGroups	14%	92%
Average	**48%**	**83%**
Deviation	**32%**	**13%**

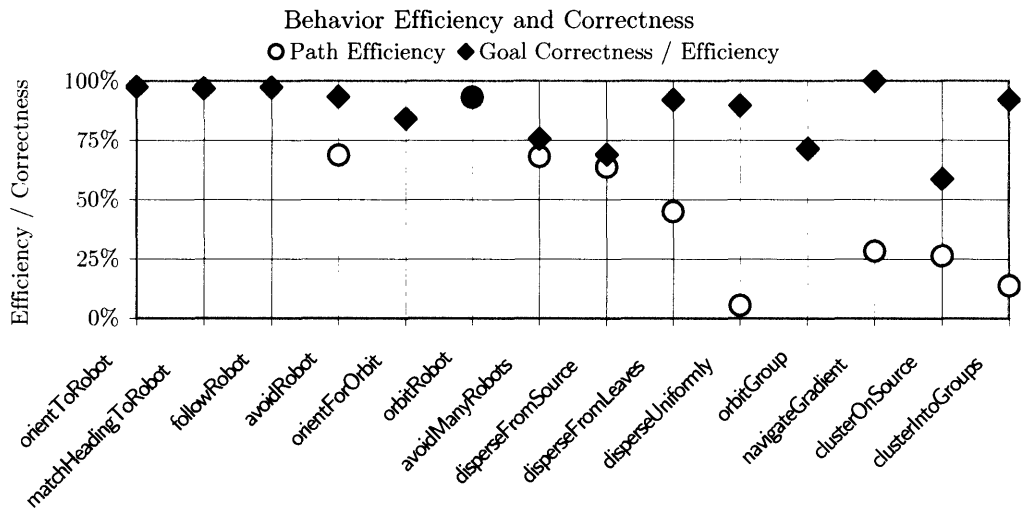


Table 6: Summary of behavior performance.

The performance of most behaviors is good, with all better than 50% accurate. This implies that given enough time, the swarm will tend to converge to the desired result. This is one of the key advantages of distributed systems compared to centralized systems. Even if individual agents do not perform correctly all the time, the system as a whole can still converge onto the correct solution.

Chapter 5.
Applications and Demonstrations

The goal of the gradient communications system and the behavior library is to provide a set of reusable algorithms and behaviors that can be used for any number of applications. At the time of this writing, there have been many, four of which are discussed below.

5.1 Surround Object

The ability to surround a phenomenon autonomously gives the swarm a way to characterize the phenomenon, or protect it from intruders. This could be useful in surrounding a chemical spill, or mapping out their environment.

Spec

- Form a perimeter of uniformly spaced robots around a designated object.
- Robots should move along the shortest path to final position at constant velocity

Implementation

The behavior starts when one robot finds the appropriate sensory stimulus in its environment and becomes the source for the "object" gradient. In our example, the bump sensors are used to detect the phenomena - any object that is not a robot is fit to be surrounded. The `stopOnBump` behavior causes each robot to stop if it detects an obstacle that is not a robot. The first robot becomes the source for the object gradient, and uses one of the data fields to broadcast that it is robot number one in the perimeter. It flashes its green light to indicate that it is the first robot in the chain.

Any other robot that can communicate with the first robot uses the `orbitGroup` behavior until it also collides with an object that is not a robot. In the example shown in Figure 63, the robots are orbiting counterclockwise. It becomes the second robot in the perimeter, and flashes its yellow light. This process continues until an orbiting

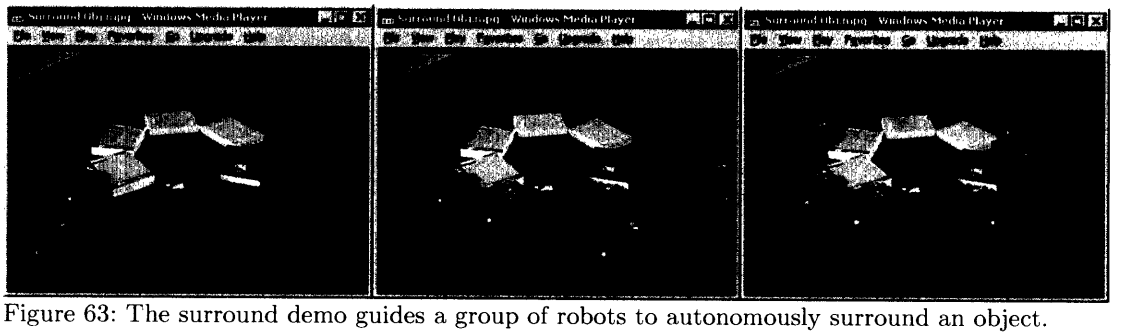


Figure 63: The surround demo guides a group of robots to autonomously surround an object.

robot comes within one orbit radius of the first robot. This means that there is not enough room for another orbiter, so this robot is now the last robot. It flashes its red light to indicate that it is the last robot in the perimeter, and becomes a source for a gradient that broadcasts the total number of robots in the perimeter. The perimeter can be approximated by multiplying the total number of robots by the `orbitGroup` radius

This was an early demo application, and it was developed without the aid of the `countingGradients` or the `navigateGradient` behavior, These more sophisticated behaviors could increase the efficiency and robustness of this demo significantly by providing reliable navigation to open perimeter points, and by eliminating the need for robots to explicitly count their way around the perimeter. This would make it less brittle if a robot is removed from the middle of the perimeter, after it has found the object.

5.2 The MegaDemo

The MegaDemo is a showcase for the swarm behavior components. It is designed to be a human-operated, visually interesting demo. One to three users control distinguished robots via remote control – the red leader, the yellow leader, and the green leader. (Also called Rhindle, Yorgle, and Grundle, respectively).

The leaders tessellate the swarm with normal gradients as shown in Figure 9. Other robots, called "minion robots", pledge allegiance to the leader they are closest to, and begin flashing the appropriate color LED. Ties between hop counts are decided based on leader color, with the ordering of red > green > yellow. Robots that are not part of a leader's connected component go into an idle mode and stay still.

The active behavior for each leader's subgroup is selected by the user with the remote control. There are eight behaviors to choose from:

1. `followTheLeader` (parade length)
2. `clusterOnGradientSource` (separation distance)
3. `disperseFromGradientSource` (separation distance)
4. `intersticialNavigation`
5. `clusterIntoGroups` (number of groups)
6. `matchOrientation`
7. `orbitRobot` (orbit radius)
8. `groupPowerDown`

Some behaviors have parameters that can be dynamically tuned by the user during operation. These are indicated in parenthesis after the behavior

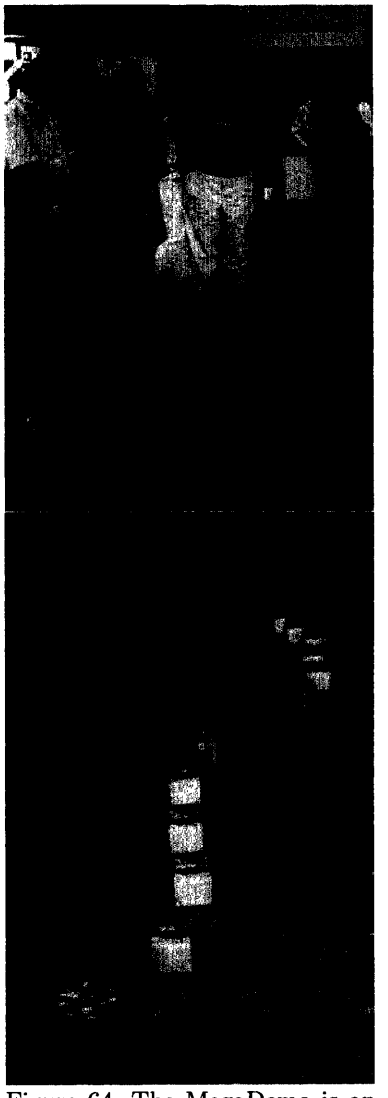


Figure 64: The MegaDemo is an interactive program that allows one to three users to control the swarm and display basic behaviors. The top picture shows the author at the helm, (In the middle), and the bottom is the crowd pleasing `followTheLeader` behavior.

name. Each behavior is heralded by a distinguishing song from the group leader the first time it is executes. After that, the entire group plays a note from the C Major scale that indicates the selected behavior.

This program is kept up-to-date as new behaviors and distributed algorithms are added to the Swarm's repertoire. Having a direct human interaction with the demo is useful for dynamically responding to environmental conditions or behavior requests, but can confuse people into thinking that centralized control is the main goal of the research.

5.3 Lemmings

The Lemelson Center at the Smithsonian Museum of American History invited the Swarm to participate in their "Toy Invention Festival". Wanting to present a more interactive exhibit than the MegaDemo, I designed the Lemmings program. The namesake is a video game from the early '80s that challenged users to guide a group of lovable, but cerebrally challenged, on-screen protagonists to escape from various mazes. The concept translates well to the swarm, with the goal being to get a group of lovable, but cerebrally challenged, robots to escape from a maze. In addition, the demo has to be accessible to a broad audience, as young children would be the primary users.

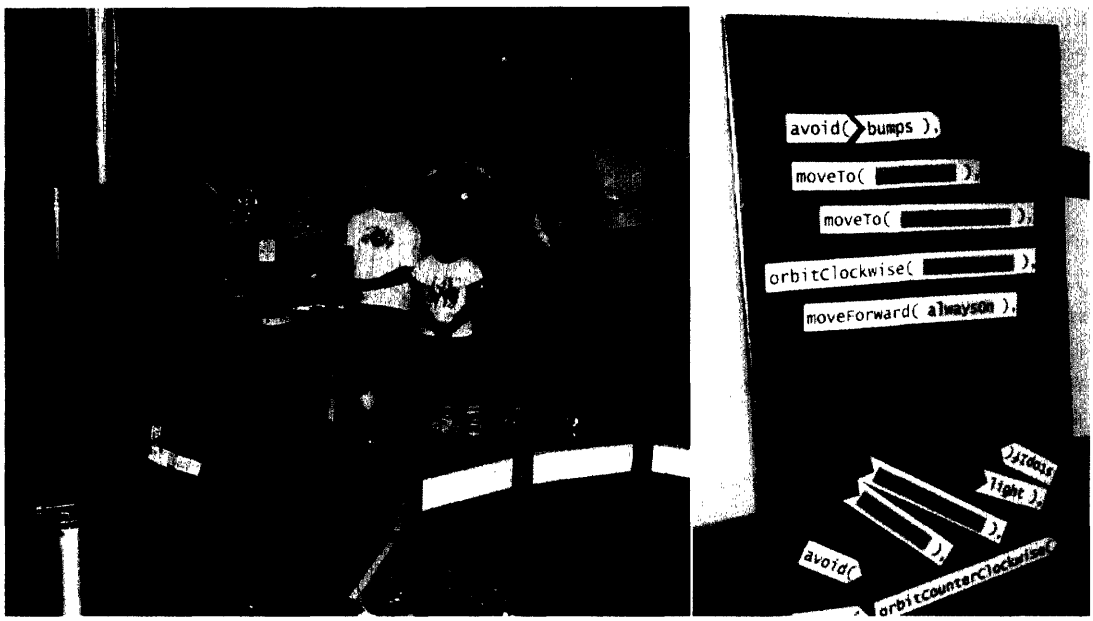


Figure 65: The Lemmings Language lets younger robotisists program the Swarm. The goal is to get a group robots to escape from a maze. The smaller programmers use the magnetic language pieces shown in the left hand picture to pair behaviors with sensory inputs, which forms a program for the Swarm. The larger programmer transcribes the code from the magnets, compiles it, and downloads it to the swarm. The lemmings interpreter executes the first active behavior , starting from the top of the list, which creates a prioritization of behaviors – the top ones are more important than those lower on the list. A good time is had by all!

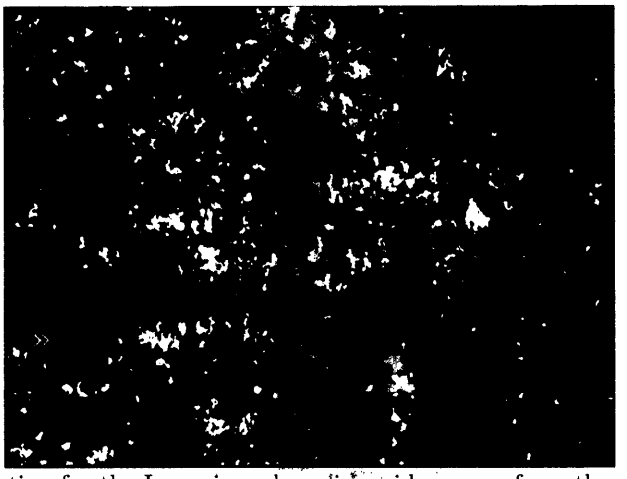


Figure 66: The inspiration for the Lemmings demo is a video game from the early 80's..

There are three types of robots in the Lemmings game: the goal robot, beacon robots, and lemming robots. The goal and beacon robots are marked by flags, and each is the source for a corresponding gradient. The user programs the lemming robots with a subset of the swarm behavior library. The set of behaviors and sensors used by the demo is shown in Figure 68.

Smaller programmers use the magnetic language pieces shown in the right picture of Figure 65 to pair behaviors with sensory inputs, which forms a program for the Swarm. The larger programmer transcribes the code from the magnets, compiles it, and downloads it to the swarm. The lemmings interpreter start from the top of the program ad executes the first behavior that has an active sensor. This creates a prioritization of behaviors – the top ones will override, or subsume, those that are lower on the list.

The right hand picture in Figure 66 shows an example Swarm lemmings maze. One possible solution program for this maze is given in Figure 67. The avoid(bumps) behavior is almost always the highest priority behavior, as obstacles can seldom be

```
/******** Include Files ********/
#include "swarmOS.h"
#include "neighborSystem.h"
#include "behaviorSystem.h"
#include "lemmings.h"

behaviorListStruct lemmingBehaviorProgram[] = {
        /* Put your software here, in order of priority.  Highest priority is first. */
        avoid(bumps),
        moveTo(goalRobot),
        moveTo(goalGradient),
        orbitCounterClockwise(beaconRobot),
        moveForward(alwaysOn),
};
```

Figure 67: A sample program for the Lemmings demo. The prioritization of these behaviors guides the robots around the beacon and towards the goal.

ignored. If a lemming robot can detect the goal robot with `moveTo(goalRobot)`, it will move towards it. Failing that, the `moveTo(goalGradient)` behaviors can provide some longer-range navigation. The `orbitCounterClockwise(beaconRobot)` behavior will slingshot the lemmings around the beacon robot and put them within range of the goal, while the `moveForward(alwaysOn)` behavior will prevent the robots from standing still.

Since its creation, the lemmings demo has been given almost ten times, to audiences ranging in abilities from 4th grade to seasoned engineers. The problem-solving techniques needed to understand how half-dozen robots can solve the problem challenges different groups in different ways. Inexperienced programmers and children don't understand why `avoid(bumps)` should be the highest priority, while engineers claim they can't solve the problem without some kind of conditional operators. Overall, the program has been very well received, and serves as a useful tool for introducing distributed algorithms and behavior-based programming to children of all ages.

5.4 The Swarm Choir

There are many important and useful applications for swarms of autonomous robots. Playing music in a robotic choir is not one of these applications, but serves as a proving ground for a `temporalSync` algorithm that is not described in this work, and the `clusterIntoGroups` behavior from section 4.5.9. Developed for this demo, these two behaviors can be used to construct more serious applications. Each SwarmBot has a 1.1 watt audio system capable of playing MIDI files, the behaviors coordinate the robots motion and timing.

There are three phases to the demo. Currently, each step is mediated by a human operator, but the algorithms and behaviors will work without human intervention.

1. The robots are told what musical selection they are going to play. Each robot picks an instrument to play from ones used in the piece of music.
2. A leader is elected using a gradient with lateral inhibition. All other robot synchronize with respect to this leader. This allows them to play in time.
3. the `clusterIntoGroups` behavior moves robots that are playing the same instruments into groups. This creates a pleasing visual and aural experience for the user and the audience.

The net result is a swarm of robots playing music together, spatially organized such that robots playing the same instrument are near each other, as shown in Figure 68. The static nature of `clusterIntoGroups` is sub optimal for this demo. Because group population ratios cannot be specified and robots leave to go charge,

Figure 68: The Swarm Choir performs at the iRobot Holiday party.

proper instrumentation cannot be specified or maintained. A more dynamic clusterIntoGroups behavior would correct these problems.

5.5 Directed Dispersion

Almost every application for swarms of robots requires them to disperse throughout their environment. Exploration, surveillance, and security applications all require coverage of large areas. The directedDispersion algorithm is designed to disperse a large swarm of robots into an enclosed space quickly and efficiently.

In order for a dispersion algorithm to be effective on a swarm of physical robots, it must take into account engineering concerns: allowing for robot and communications failures and maintaining network connectivity, especially between the swarm and the chargers.

The goal of the directedDispersion algorithm is to spread robots throughout an enclosed space quickly and uniformly, while keeping each robot connected to the network and ensuring a gradient communications route back to the chargers. The dispersion is accomplished by using two algorithms that alternate running on the swarm: diperseUniformly and frontierGuidedDispersion, which is based on disperseFromLeaves.

The diperseUniformly algorithm from section 4.5.4 is responsible for spreading the robots evenly throughout their environment, using naturally occurring walls and the maximum dispersion distance of r_{safe} as boundary conditions. The frontierGuidedDispersion algorithm directs robots towards unexplored areas, and is designed to perform well both in open environments and in environments with constrictions and complex layouts.

Frontier Determination

Robots need to identify their positions in the graph as: frontier, wall, or interior. "Frontier" robots are on the edge of explored space, and are used to guide the swarm into new areas. "Wall" robots are those that detect an obstacle with the ISIS system. The remainder are "interior" robots, as illustrated in Figure 69. The detectEdge algorithm is used for part of this determination.

```
frontierDetermination() returns integer
1.  if (detectEdge())
2.    graphPosition ⇐ FRONTIERROBOT
3.  else if radar.range < WALLRANGE
4.    graphPosition ⇐ WALLROBOT
5.  else
6.    graphPosition ⇐ INTERIORROBOT
7.  endif
8.  return graphPosition
```

Frontier Communication and Swarm Motion

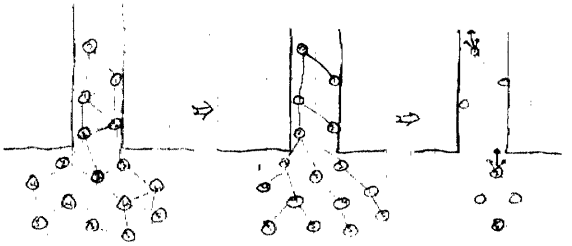


Figure 69: Frontier robots guide the swarm into unexplored areas. First, a robot nominates itself as a frontier. Then a gradient propagates throughout the network, alerting all other robots that a frontier has been found and forming a tree rooted at the frontier robot. All robots then move away from their children in this tree. Leaves on the tree do not move, allowing previously dispersed robots to remain stationary.

Once there are frontier robots active in the network, they source a gradient message to inform the rest of the swarm. The gradient trees from these sources are used to guide the swarm towards the frontier robots. Using `clusterOnSource` proved ineffective, because any algorithm that is based on clustering robots over multiple hops can cause newly discovered frontiers to pull robots away from previously explored areas. This causes a frontier to re-appear at the old location and pull the swarm back, causing oscillations, or fracturing the swarm and disconnecting robots from the chargers. In addition, follow algorithms have to be written carefully to ensure a min-cut that is greater than two. This redundancy in the communications is important to produce a robust network.

The `frontierGuidedDispersion` algorithm uses the `disperseFromLeaves` behavior to switch the focal point in gradient-based navigation from the source of the gradient tree to the leaves. Robots move away from their children in the frontier tree that are closer than $\mathbf{r_{safe}}$. In order to build a reliable network, robots are not allowed to move unless they are in contact with at least two children in the frontier tree to disperse from. This increases the min-cut of the network to two while the robots are dispersing, which is essential for reliable communications when gaps can be created by corners or robots heading home to charge.

The properties of `disperseFromLeaves` behavior has the leaf robots remain stationary while the rest of the swarm moves away from them. This ensures that robots are left behind to provide a route to the chargers, and that once an area has been explored, another frontier will not be able to pull the leaf robots or their parents out of that area. Essentially, the leaves become "anchors" and then limit the dispersion of robots away from them to a distance of $\mathbf{r_{safe}}$.[7] As robots move away from the leaves, they move closer to their upstream robots, causing a chain reaction that eventually moves all the robots towards the frontiers.

Multiple frontiers often form as the Swarm explores the environment. Their gradients tessellate the swarm based on hop count as shown in Figure 9. This is useful because progress of distant frontiers will be slowed as interior robots disperse towards frontiers with smaller hop counts, allowing these closer frontiers to catch up. This tends to make the swarm explore the entire building in a breadth-first fashion.

[7] Another way to think about this is to imagine that any robot that is not maximally dispersed from its children will head towards the frontier, causing its parent to move towards the frontier, etc. This results in a "wave" of motion that the frontier "surfs" forward .

```
frontierGuidedDispersion(beh)
1.  childNbrSet ⇐ nbrOp(nbr.M[FRONTIERGRADIENTTYPE].hops >
                                        self.M[FRONTIERGRADIENTTYPE].hops)
2.  if size(childNbrList) > 2
3.    disperseFromLeaves(beh, FRONTIERGRADIENTTYPE, RSAFE)
4.  endif
```

Line 1 creates a set of robots that are children in the frontier gradient tree. If there are more than two children robots, then this robot can disperse. This helps provide a min-cut of the graph of no less than two, which is critical for network robustness.

Putting it Together: directedDispersion

The frontierDetermination and frontierGuidedDispersion behaviors are combined into directedDispersion:

```
directedDispersion(beh)
1.  if frontierDetermination() = FRONTIERROBOT
2.    gradientSource(self.M[FRONTIERGRADIENTTYPE], NORMAL)
3.  endif

4.  if self.M[FRONTIERGRADIENTTYPE].isActive = TRUE
5.    frontierGuidedDispersion(beh)
6.  else
7.    disperseUniformly(beh)
8.  endif
```

Line 1 determines the robots position in the network. If the robot is a frontier, line 2 sources a frontier gradient. Line 4 checks to see if there is a gradient present in the network. If so, then line 5 uses **disperseFromLeaves** to disperse the swarm into the environment. If there is no frontier gradient, then line 7 uses **disperseUniformly** to equalize the positions of the swarm.

The "pressure" from **diperseUniformly** tends to push robots into open spaces and tight constrictions. Eventually, a frontier is formed and its gradient messages activate the **frontierGuidedDispersion** behavior, which causes a directed dispersion towards the frontiers. This behavior stays active until all frontiers encounter walls or move to the interior of the swarm. Termination of the combined algorithm is defined when the frontier behavior stays inactive for a specified amount of time. Unfortunately, complex environments, sensor noise, and robots leaving to charge can make it difficult to quantify this time. We use ten seconds for the experimental results.

Experimental Results

Fifty-six robots were used with a reduced ISIS communications power setting to explore a small office-like environment with three goals placed as shown in Figure 70. The swarm was released and times to reach the three goals and full dispersion were recorded. Five algorithms were compared.

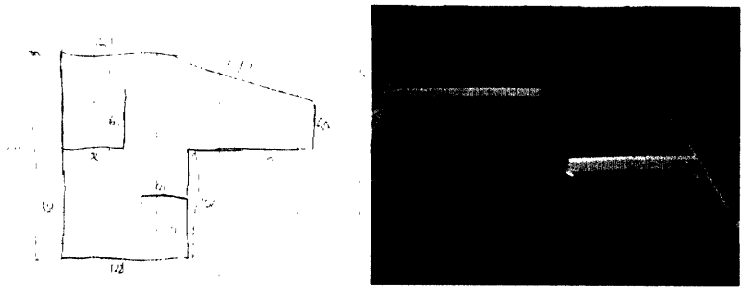


Figure 70: **Left:** The dispersion algorithms were tested in a small office-like environment. Goals were placed at the locations shown, and robots were released from the area at the bottom. **Right:** An example dispersion into the test space.

`idealGasMotion`: Robots move in straight lines unless they collide with each other or with a wall. The network often breaks into disconnected components. Inter-robot interference is a problem, with robots colliding often. There is no termination condition, and dispersion is rarely uniform.

`disperseFromSource`: Described in section 0. Network connectivity is maintained during the dispersion process if $r_{disperse} \leq r_{safe}$. Uniform, complete coverage only occurs if the environment area is known in advance and $r_{disperse}$ is selected accordingly, otherwise robots will either bunch up at boundaries or not fill the area. However, the dispersion is very efficient, quickly reaching all goals and full dispersion.

`avoidClosestNeighbor`: Robots move away from their closest neighbor at constant velocity if $r < r_{disperse}$. Network connectivity can be maintained if $r_{disperse} \leq r_{safe}$. There is no termination condition. This is very similar to `diperseUniformly`, and the results are also similar. Dispersion is uniform, but robots oscillate back and forth between closest neighbors.

`diperseUniformly`: Described in section 4.5.4. This algorithm runs slower than avoid-closest-neighbor, but the motion is smoother. It has very uniform dispersion and maintains network connectivity. Robots remain stationary after dispersion

`directedDispersion`: Described above. The robots rarely head in the wrong direction, and effectively push frontiers to the boundaries. The algorithm terminates with uniform coverage and robots remain fairly stationary after dispersion

Additional tests were conducted in a empty schoolhouse. A total of 108 robots were able to effectively disperse into about 3000 ft^2 of indoor space, locate an object of interest, and lead a human to it. Multiple arrangements of rooms were tried, with several constrictions, sharp turns, and large open areas. The robots ran almost continuously for six hours, returning to charge when needed, and filling gaps in the

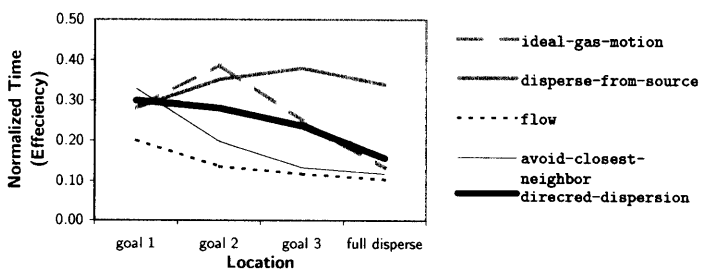


Figure 71: Dispersion efficiencies of the five algorithms tested.

dispersion when required.

Conclusion

The `directedDispersion` behavior allows robots to explore large, complex, indoor environments. Multiple frontiers create a structured communication network that the robots can use for navigation into unexplored areas of the environment. Dispersion tends to occur in a breadth-first fashion. Gradient message clean-up is important as frontier gradient sources start and stop sourcing to maintain the structure of the dispersion. Practical dispersion algorithms can be designed to meet efficiency, robustness, scalability, and correctness constraints.

5.6 Summary

Many applications have been constructed from the Swarm Behavior Library. The behaviors and design philosophy of developing and testing software on the robots directly has proven to be an effective way to develop applications. The final state of these applications is often the result of testing dozens of different behavior variations and combinations. Although the system is still not as efficient as a software simulation, this disadvantage is more than compensated by the richness of the "hardware simulation" - real robots in actual environments.

This approach is expeditious and flexible, it is possible to combine behaviors quickly and get predictable group actions. For example, the autonomous charging behaviors (not described in detail here) use internal measurements on individual robots to decide when to recharge, then each robot uses the `navigationGradient` behavior to move towards the charging gradient and dock with their chargers. This runs in conjunction with other behaviors, but does not affect the performance of the group as a whole.

However, this approach provides the developer with only a veneer between the desired application and the complex interactions of multiple robots. Behaviors running simultaneously can have unpredictable interactions. Many of the applications developed avoid this problem by forcing behaviors to be temporally mutually exclusive. For example, the Megademo and `directedDispersion` applications use gradients to switch between different modes of operation, preventing unexpected interactions. A more sophisticated set of behaviors will probably not help alleviate this problem to any great degree, a centralized development environment that understands the interactions between behaviors and the user's design goals will be required to achieve this goal.

Chapter 6.
Conclusions and Future Work

6.1 Limitations

Behavior-Based Control

The swarm algorithms exploit the advantages of behavior-based software – robots pick up their stare from their neighbors and the environment, which allows them to perform well in dynamic, unstructured environments, and allows the composition of the swarm to change over time and not affect performance. However, the problems with behavior-based control all apply: there is no planning to overcome future problems, no learning from past errors, and no map or model of the world to reason from or share with the user. Even a slightly pathological environment can defeat the entire swarm if the programmer hasn't added the requisite behaviors to handle the situation.

Development Environment

The iRobot Swarm provides excellent debugging feedback with a combination of audio cues, large status lights on each robot, remote control and downloading, and low-level debugging direct to the processor core. This development environment lowers the barriers to experimenting with new software, it can be faster to write some test code and download it to 30 robots than it is to reason through the algorithm carefully. While this is certainly not the desired approach to development, it is important that the energy barriers to working with the robots be kept as small as possible, in order to realize their ability to ground the development in reality and find false assumptions.

However, the quantitative measurements presented here on path efficiency and algorithm correctness required considerable effort to produce. The ideal swarm infrastructure would collect and maintain data from each run, with minimal user interaction. Every attempt was made to underestimate correctness and efficiency.

The Truth About Scalability

For practical reasons, scalability only works in one direction – down. It is difficult to reason through all the interactions that multiple robots will have with each other as their numbers scale from 10 to 40 to 100. An excellent example of this is the `disperseUniformly` behavior. When used with around 20 robots, the areas explored are small, ISIS power levers are small, and the behavior works well. When trying to explore larger areas with 100 robots, the ISIS power levels must be increased, which leads to the problems shown in Figure 44.

Software bugs scale non-favorably with increasing swarm sizes. A bug that occurs in every 1 out of 100 runs of a single robot can be safely ignored. On a swarm of

100 robots, this bug will be occurring 100 times more frequently, i.e. on some robot all the time, and can no longer be ignored. The bright side is that this gives you an opportunity to find bugs 100 tines as fast, although the fact that this is a benefit is often forgotten on the eve of a demo.

In general, there is some corner case, algorithmic oversight, or bug, that becomes unignorable at the next scale level. Careful development and programming will catch some of these, but research demands that we try ideas we have not yet implemented, so this will always be a problem. Good simulations can help minimize unexpected actions, but care must be taken in their use, as they often do not model the world completely.

6.2 Future Work

There is still a great unexplored research territory in distributed robotics. There are many areas for improvement on the techniques presented here, notably in the areas of path efficiency. In addition:

Counting Gradients

The counting gradient provides a lower bound on the number of robots in the network, but performs poorly with unreliable communications. A possible fix would be for each robot to keep a list of children and only count the results from new children after they have been present for p neighbor cycles. This would guarantee that results from the previous robot shave timed out.

Graph Center

Being able to computer the graph center in a distributed way would be useful in many applications. Gradients spread through the swarm fastest when sources from a robot in the center of the graph, and information can be extracted most effectively from this same robot.

Dynamic Division of Labor

The `clusterIntoGroups` behavior is very primitive and is missing two key attributes:

1. Robots should be able to adjust their group participation dynamically as needs change.
2. Robots should not have to physically move to join their group, unless that is one of the goals of forming a group.

These improvements would provide more useable dynamic task allocation for the swarm.

Axioms for Swarm Programming

The ultimate goal is to be able to program group behaviors at the group level. A Swarm Programming Language could discard the dependence on carefully engineered behaviors and provide semantics appropriate for programming

This is difficult, as there is no set of axioms for programming groups of robots.

6.3 Final Remarks

The gradient communication algorithms and behavior library work well in many applications. The gradient communications form a substrate for information sharing and

robot navigation. These algorithms were developed alongside some of the earliest behaviors, and remained unchanged for nearly all future development. Currently, there are many new algorithm ideas under development, none of which require upgrading the communication infrastructure, making it the most reused part of this work.

In general, behaviors fit into three broad categories: navigation, clustering, and dispersion. While this list is not exhaustive, it does support many applications. One of the design goals was to construct a library of reusable behaviors with predictable group results. The collection of behaviors presented here does accomplish this goal, although care must be taken in behavior assembly, even in carefully structured environments like the Lemmings demo. The Directed Dispersion Application uses `disperseUniformly` and `disperseFromLeaves` in a piecewise fashion mediated by the `detectEdges` function and the spread of a "frontier" gradient. While this does use multiple behaviors, they are mutually exclusive, and therefore do not fully demonstrate the goals of recombination and predictable interactions.

It is difficult to capture some of the design paradigms that are shared between behaviors. The static function call tree in Figure 20 shows the hierarchy of behaviors and illustrates some level of modularity and reuse. It does not show the sharing of ideas, such as vector fields, between different behaviors. Re-using these concepts is important for developing new software quickly. Overall, the algorithms work well, and can often simply be "plugged in" to a piece of software when a particular type of motion or communication is desired.

Appendices

A1. neighborOps Examples

For example, the C source to collect the set of all the neighbors around the robot looks like this:

```
neighborListStruct neighborList;

allNOp(&neighborList)
```

This populates the variable neighborList with all the neighbors of this robot. To find the closest neighbor, another operator is applied to the list:

```
neighborStruct * neighborPtr;
neighborListStruct neighborList;

neighborPtr = closestNOp(allNOp(&neighborList));
```

The closestNOp function finds the closest neighbor in the list and returns a pointer to a neighborStruct. To find the closest neighbor with the minimum hops of a gradient communication message (A surprisingly common task) the code is as follows:

```
neighborStruct * neighborPtr;
neighborListStruct neighborList;

neighborPtr =
        closestNOp(
                withGradientMinHopsNOp(
                        allNOp(&neighborList)
                , &myGradient)
        );
```

The funky indentation is helpful to see which arguments are associated with which function. The final example finds the closest neighbor with the minimum hops of either myGradient or yourGradient.

```
neighborStruct * neighborPtr;
neighborListStruct neighborList1, neighborList2, neighborList3;

neighborPtr =
        closestNOp(
                unionNOp(&neighborList3,
```

```
                    withGradientMinHopsNOp(
                          allNOp(&neighborList1),
                    &myGradient),
                    withGradientMinHopsNOp(
                          allNOp(&neighborList2),
                    &yourGradient)
              )
        );
```

Rewritten without the indentation, we have:

```
neighborStruct * neighborPtr;
neighborListStruct neighborList1, neighborList2, neighborList3;

withGradientMinHopsNOp(allNOp(&neighborList1), &myGradient);
withGradientMinHopsNOp(allNOp(&neighborList2), &yourGradient);
unionNOp(&neighborList3, &neighborList1, &neighborList2);

neighborPtr = closestNOp(&neighborList3);
```

The complete list of neighborOps and a brief description of usage is given below.

Population

`allNOp`

Returns all the neighbors that this robot can detect. This is used to populate an empty neighbor list with the current sensory data.

Set Operations

`unionNOp`

Performs a union of list1 with list2 and returns the result in list 3.

`intersectionNOp`

Performs an intersection of list1 with list2 and returns the result in list 3.

`differenceNOp`

Performs an asymmetric set difference of list1 with list2 and returns the result in list 3. e.g. {a, b, c} - {b, c, d} = {a} while {b, c, d} - {a, b, c} = {d}

`symetricDifferenceNOp`

Performs a symmetric set difference of list1 with list2 and returns the result in list 3. e.g. {a, b, c} -- {b, c, d} = {a, d} and {b, c, d} -- {a, b, c} = {d, d}

Single Neighbor Operations

`neighborStruct * anyNOp`

Returns a neighbor at random from the list.

`neighborStruct * firstNOp`

Returns the first neighbor from the list.

`neighborStruct * secondNOp`

Returns the second neighbor from the list.

`neighborStruct * closestNOp`

Returns the closest neighbor from the list.

`neighborStruct * furthestNOp`

Returns the furthest neighbor from the list.

Neighbor Properties

`withIDNOp`
: Returns a neighbor with the specified robotID.

`withHardwareTypeNOp`
: Returns neighbors of the specified hardware type. This is either SwarmBot, charger, or beacon.

`withJobNOp`
: Returns neighbors with the specified job.

`withSubJobNOp`
: Returns neighbors with the specified subjob.

`withJobModeNOp`
: Returns neighbors with the specified mode.

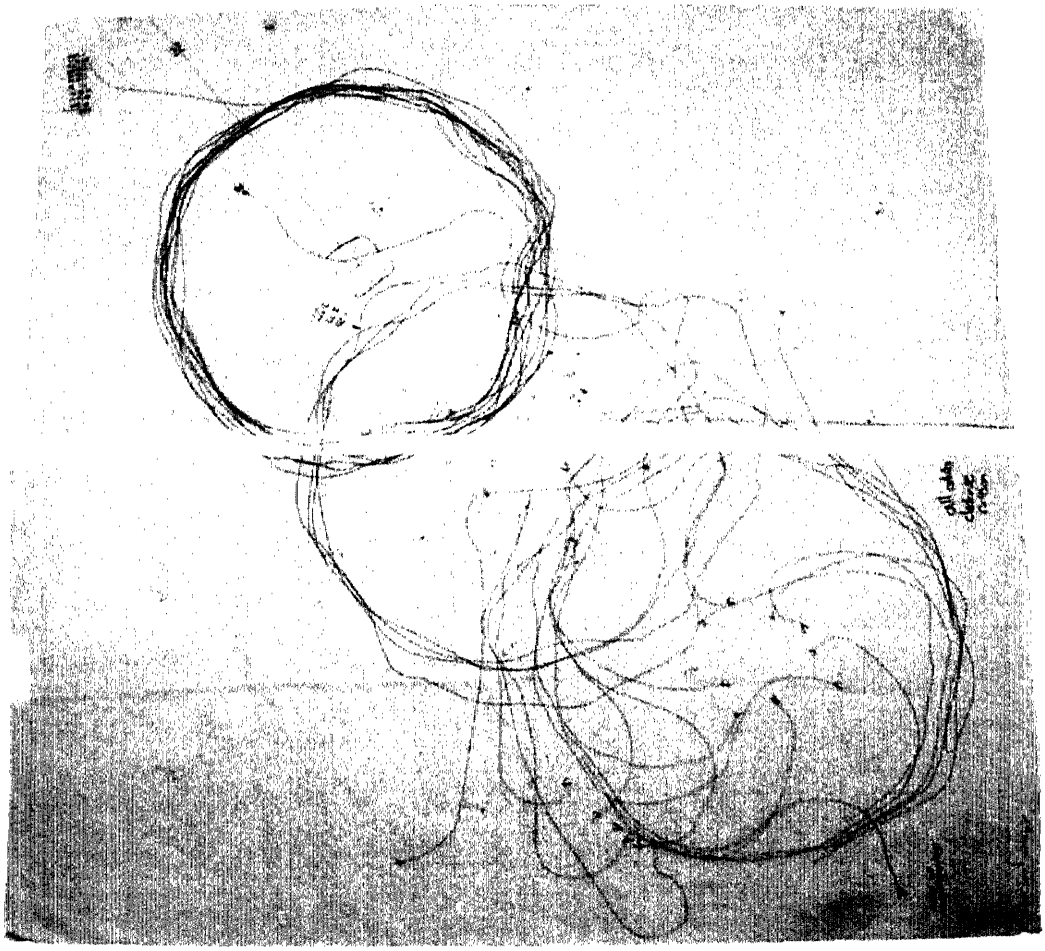
`withJobDataNOp`
: Returns neighbors with the specified data.

NeighborListOperations

`sortByRangeNOp`
: Re-sorts the neighbor list according to the each robots range.

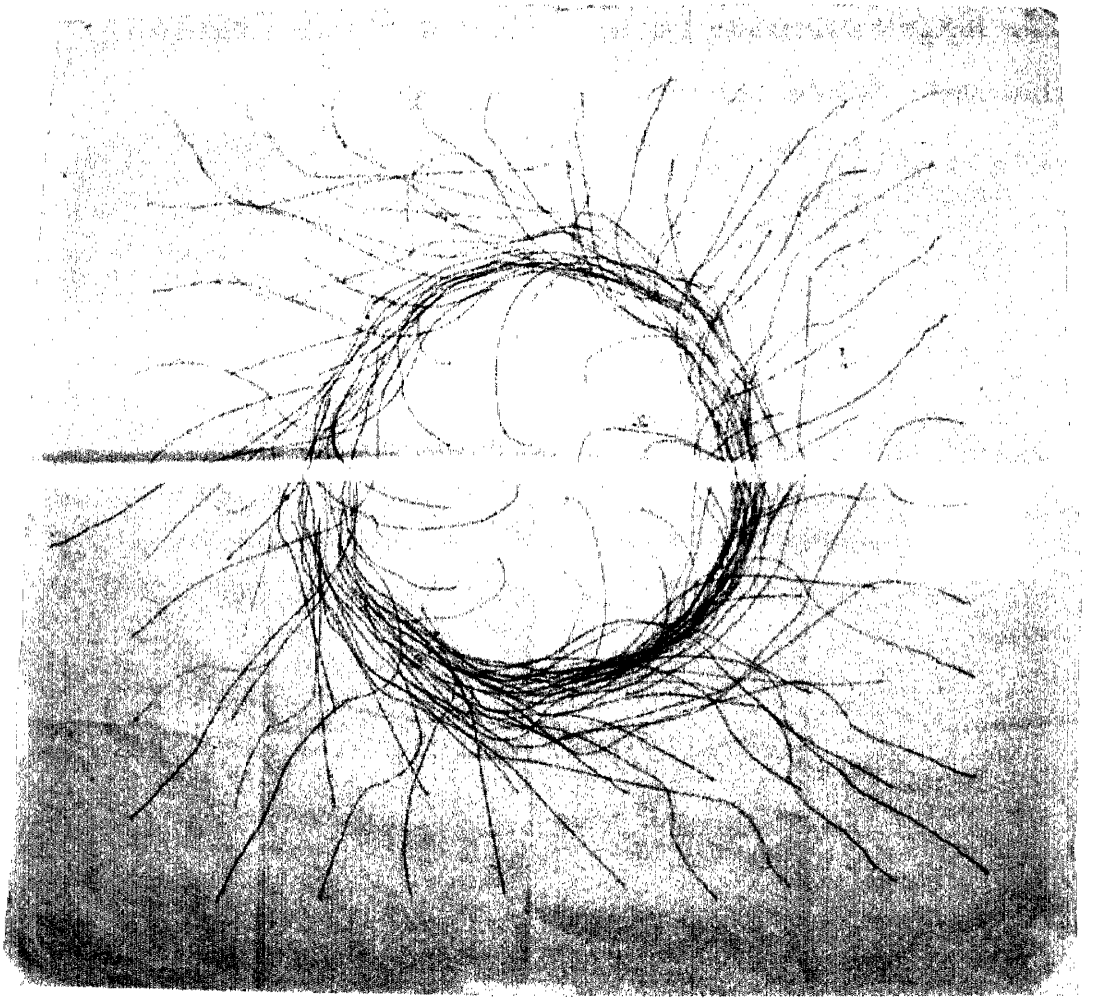
A2. Experimental Data – Robot Path Traces

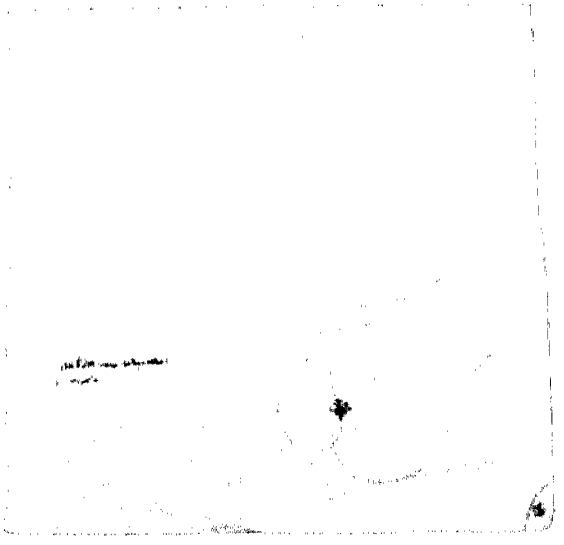
orbitRobot – Steady-state orbit, Bad orbit, Ugly flow field



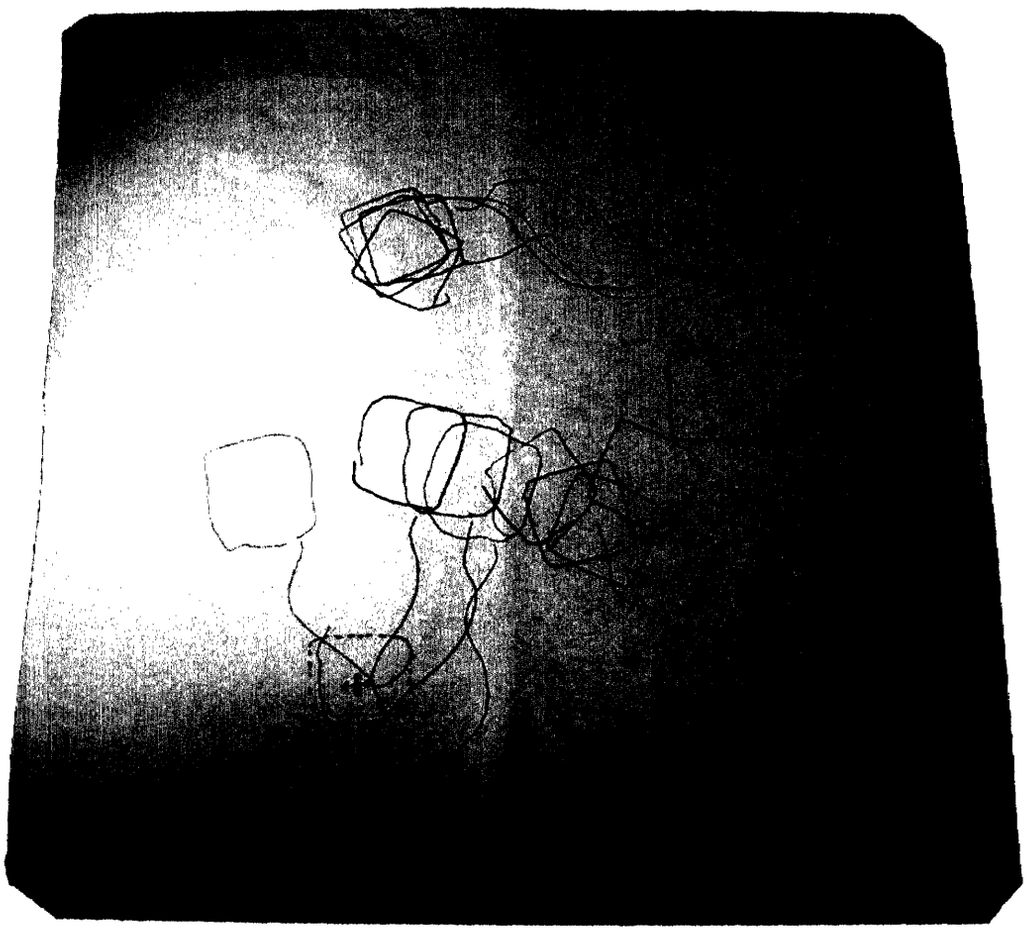
Notes:

orbitRobot – Flow Field and Orientation Field



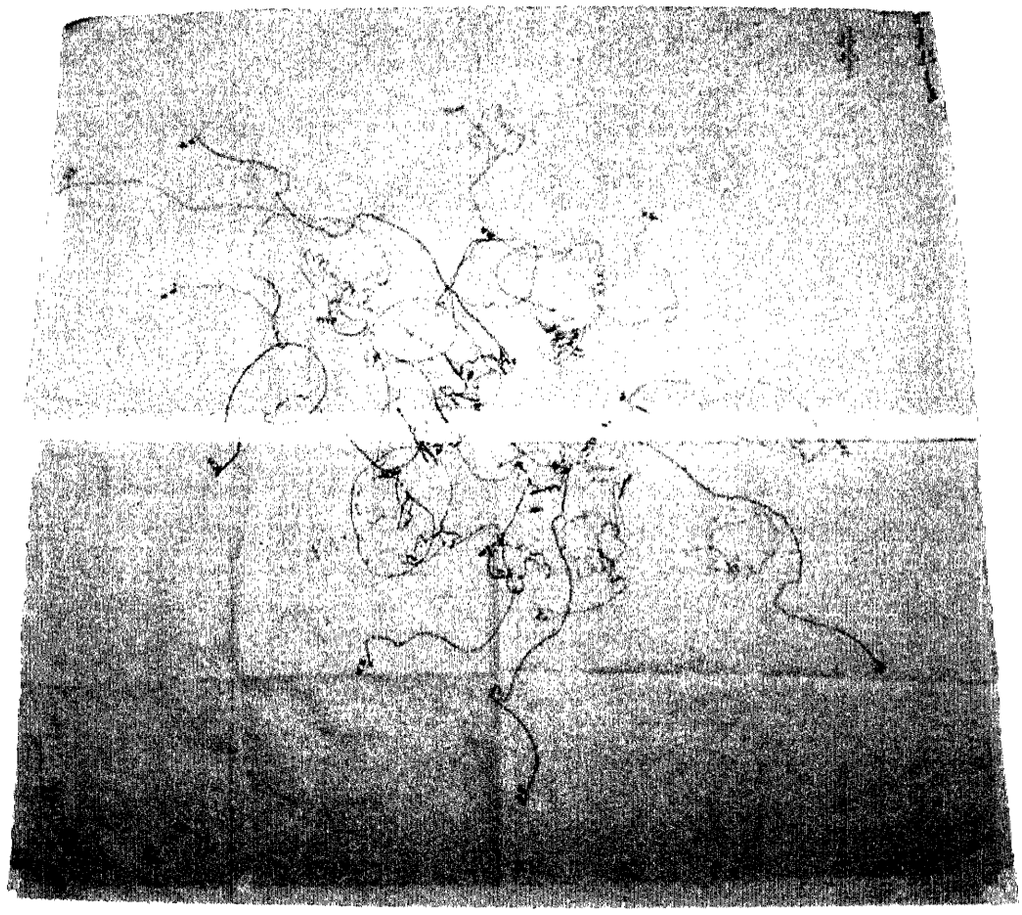


avoidManyRobots



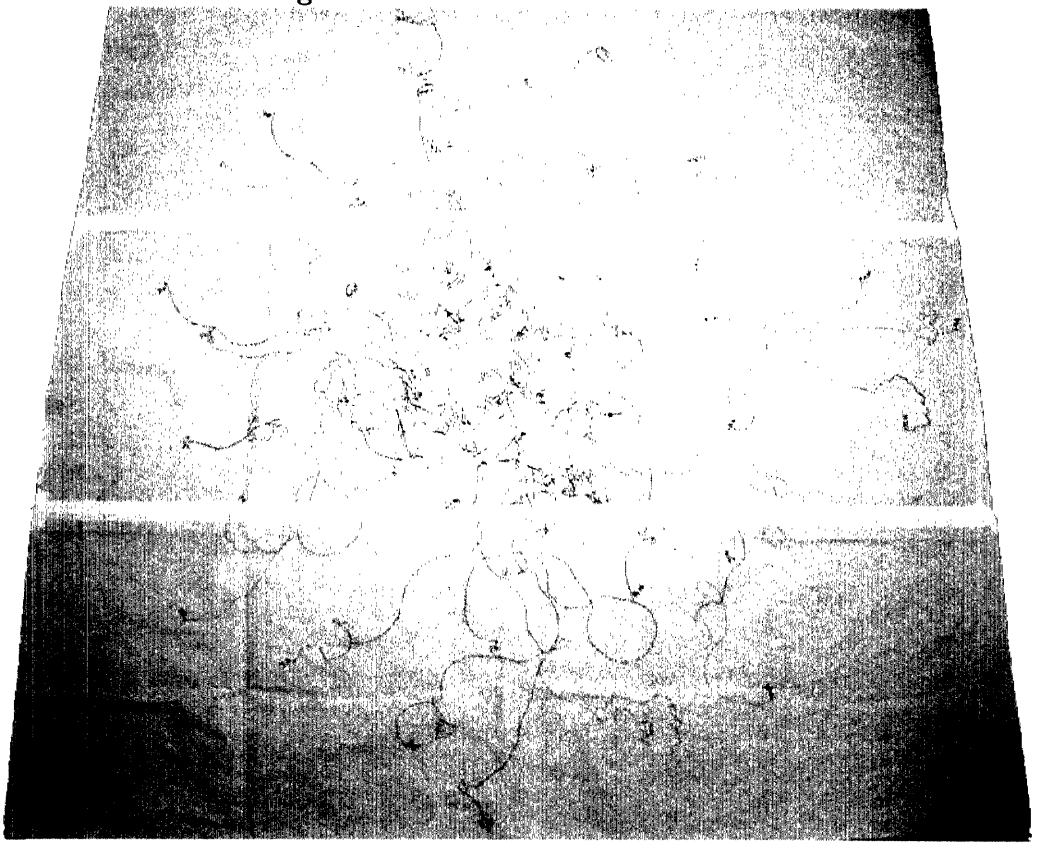
Notes:

disperseFromSource - small



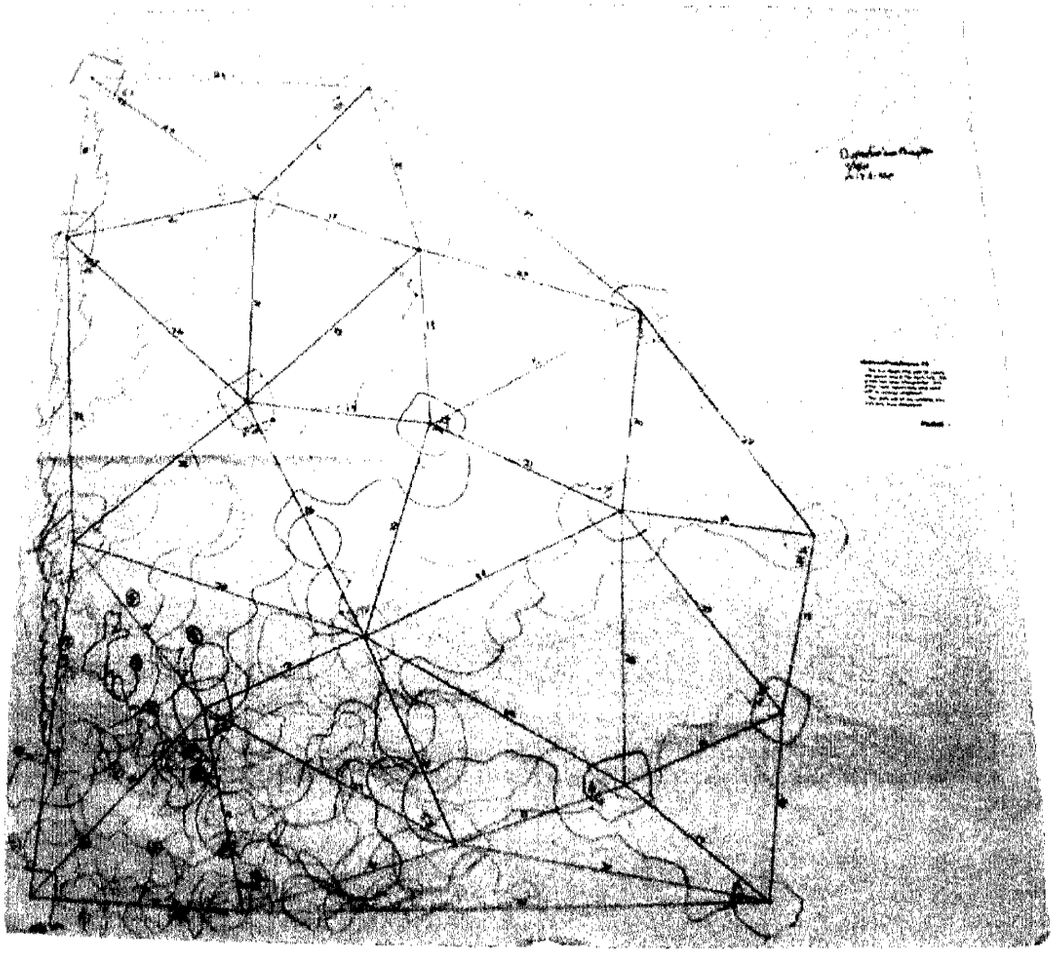
Notes:

disperseFromSource - large



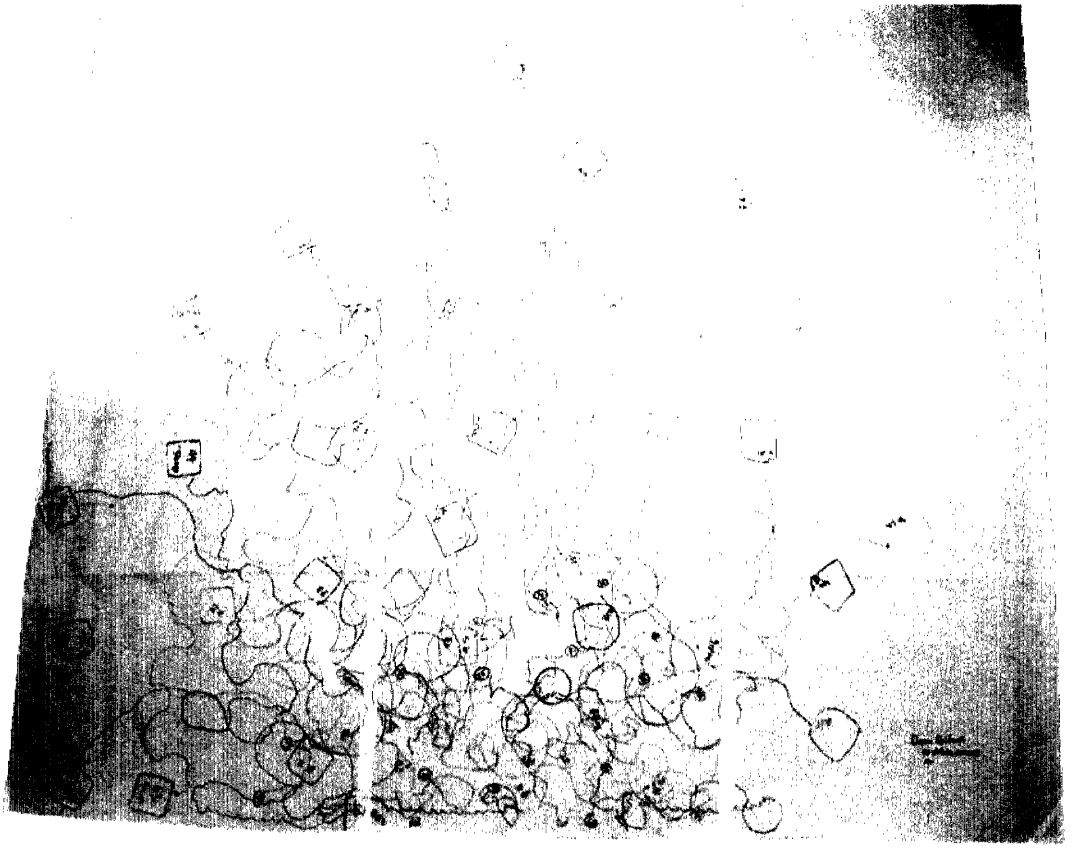
Notes:

disperseFrounSource – source in corner, triangulated



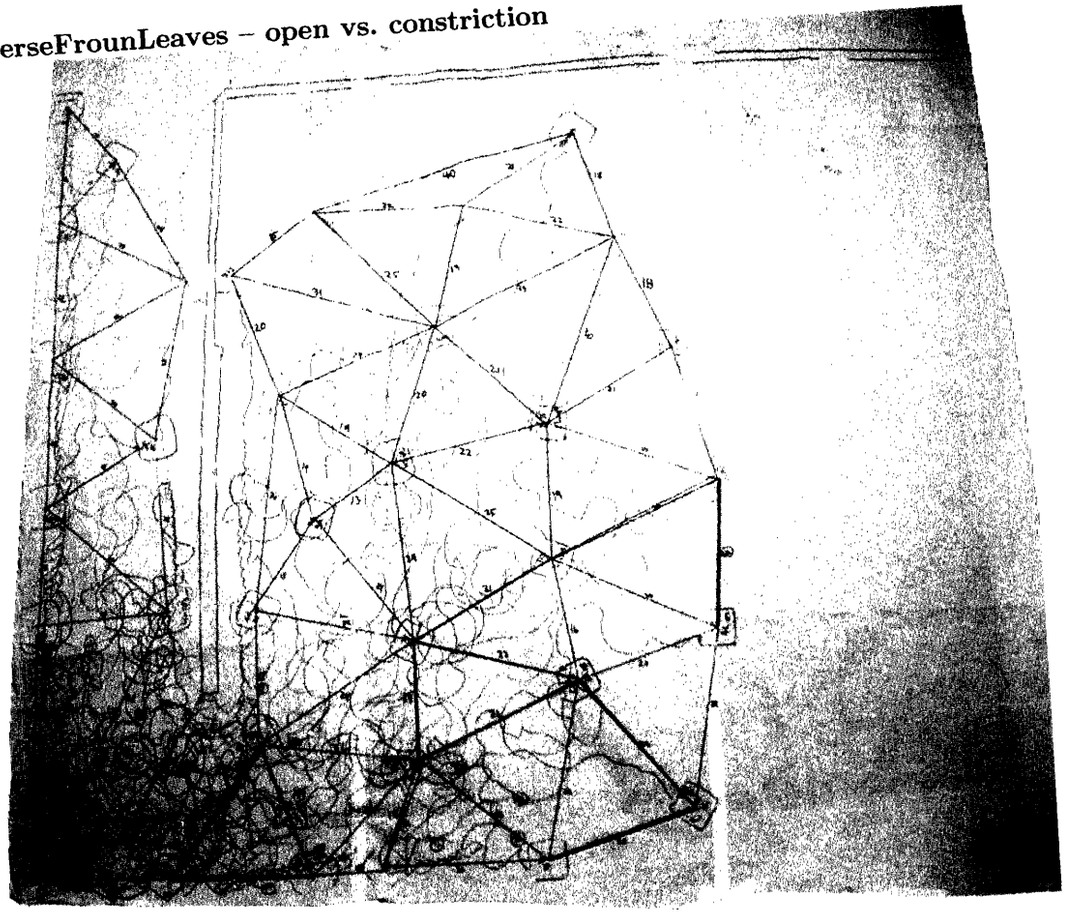
Notes:

disperseFrounLeaves – open environment



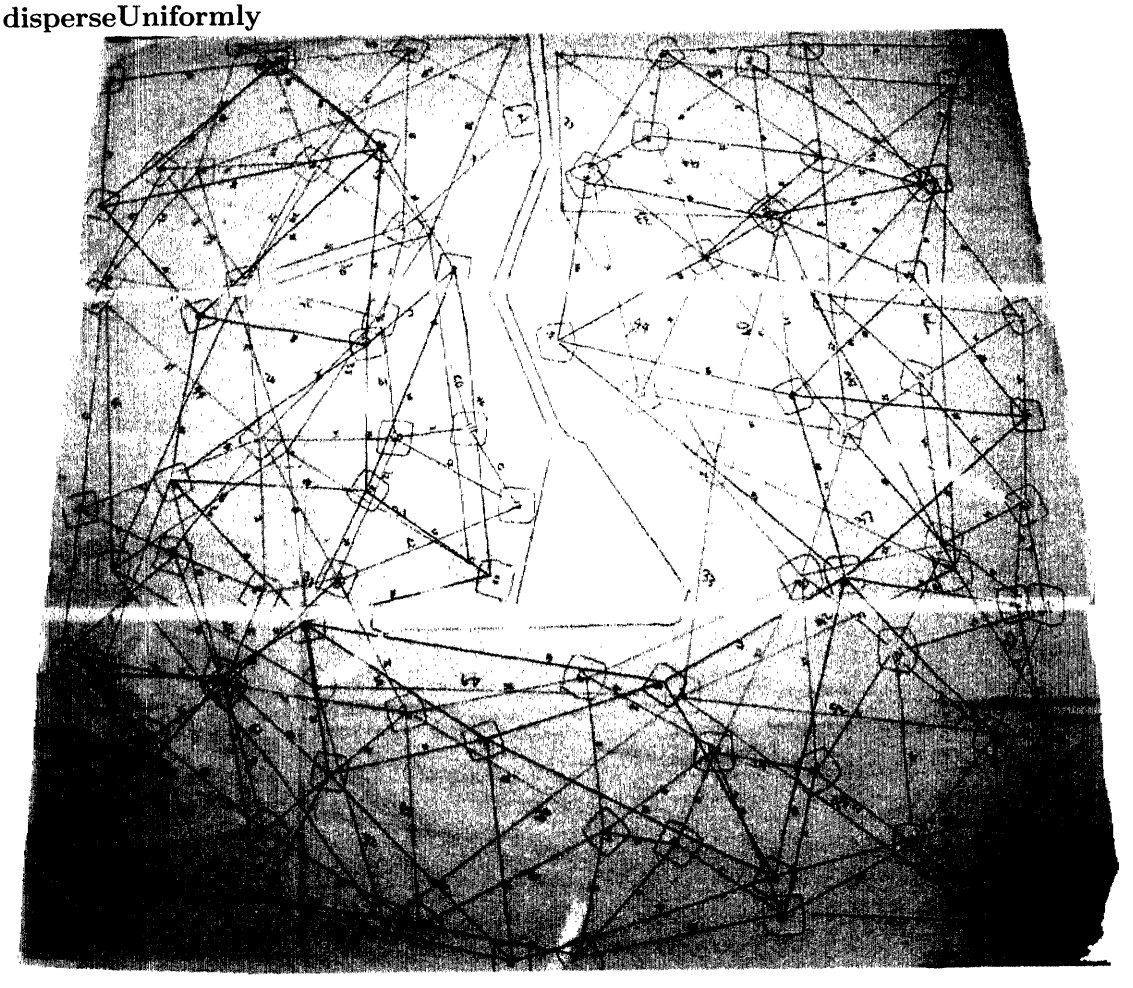
Notes:

disperseFrounLeaves – open vs. constriction



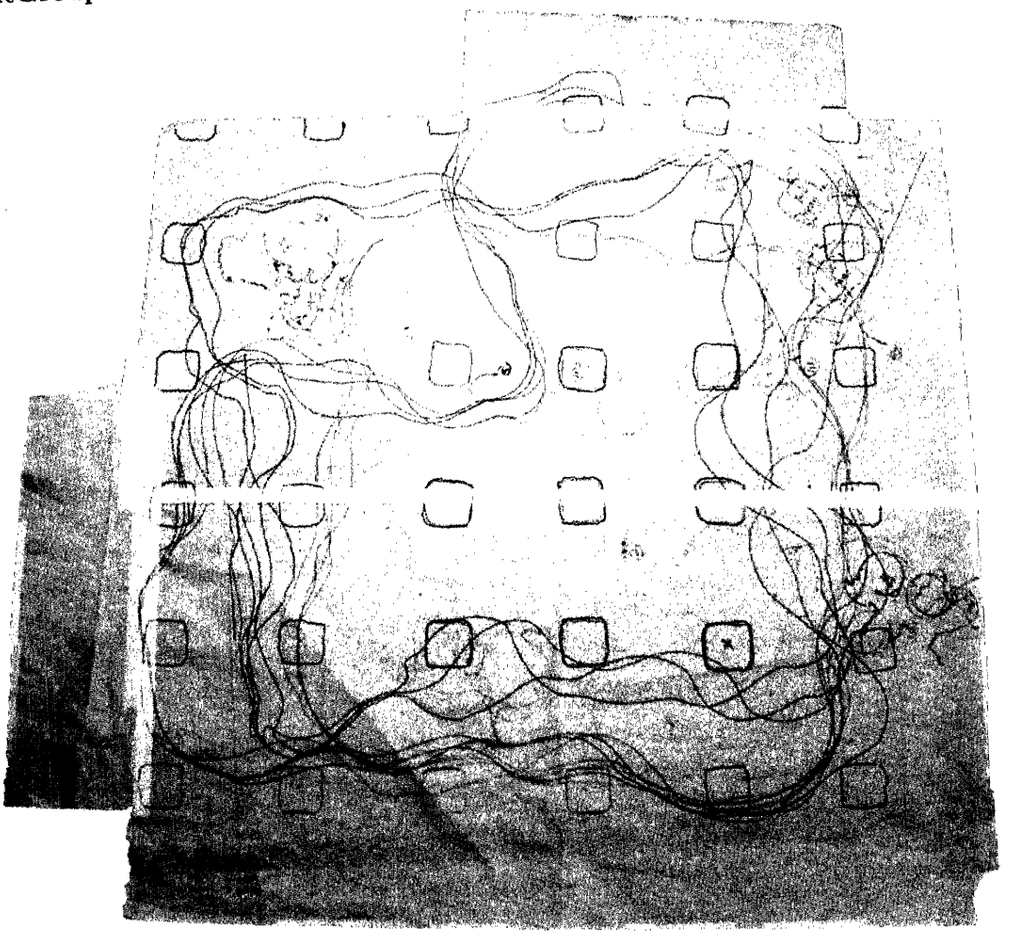
Notes:

disperseUniformly



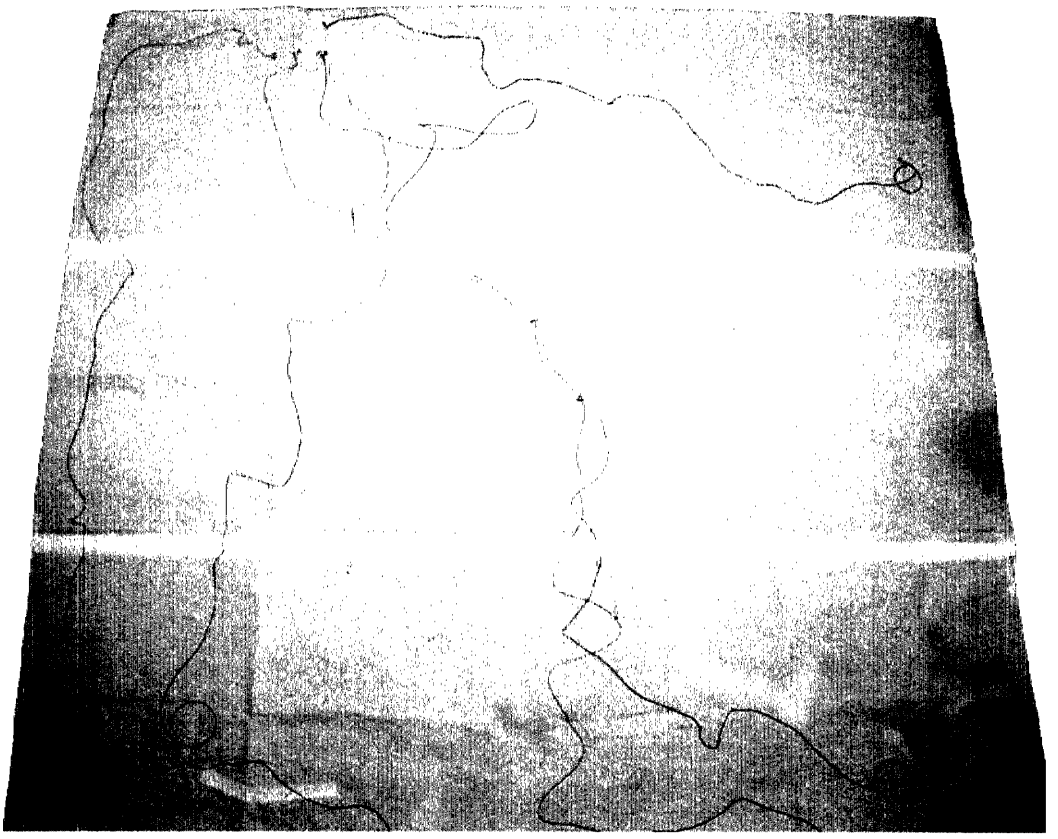
Notes:

orbitGroup



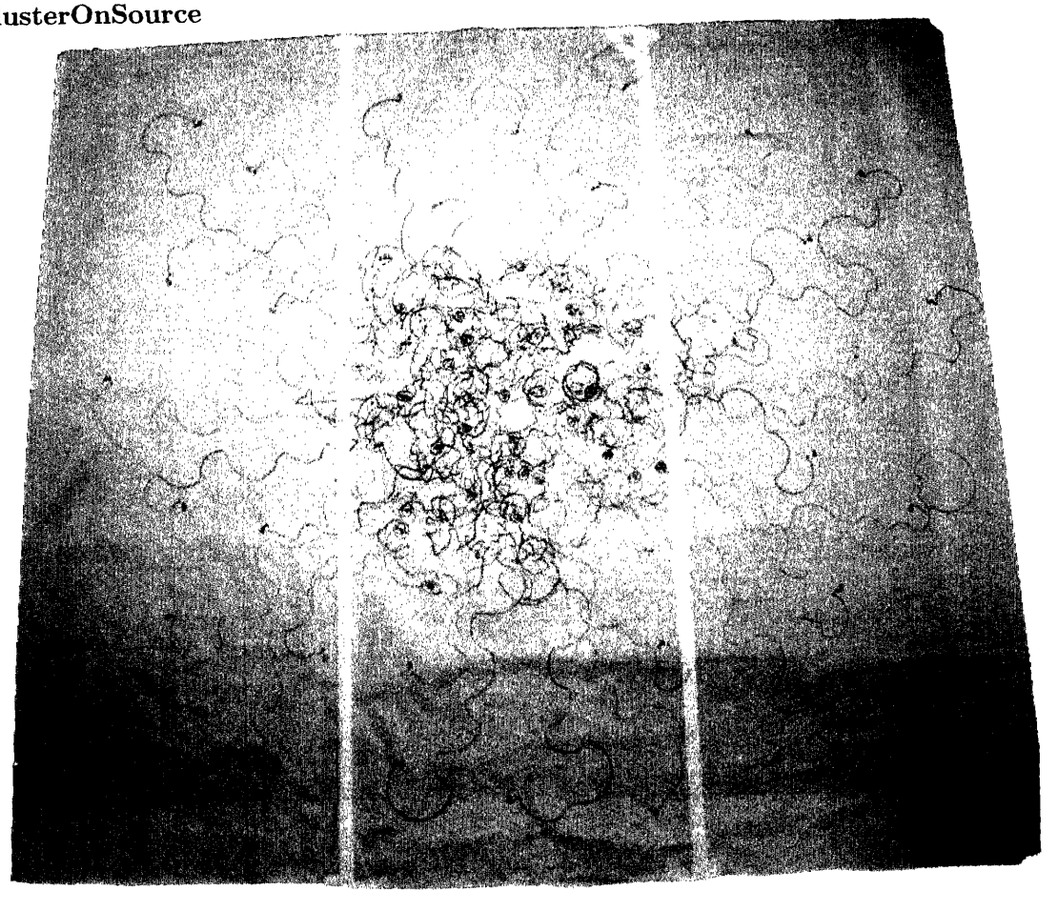
Notes:

navigateGradient



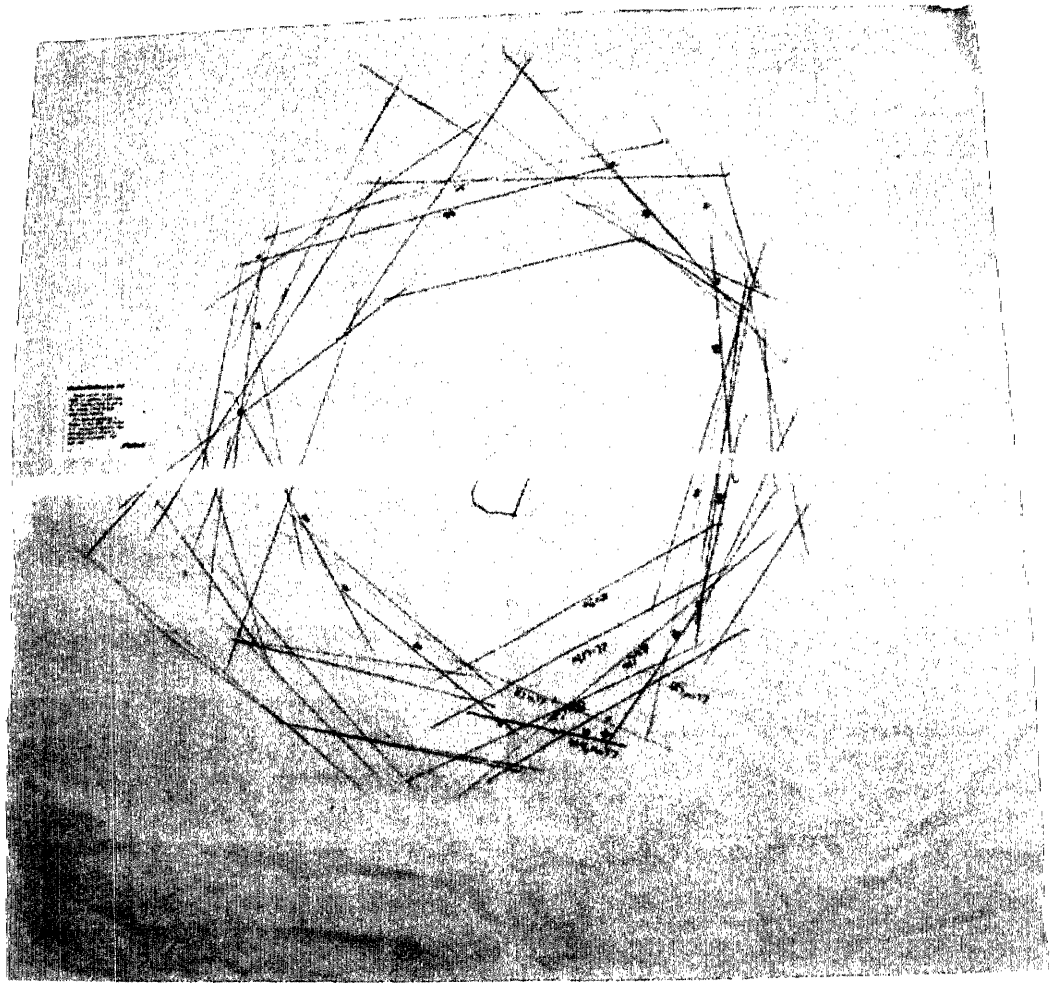
Notes:

clusterOnSource



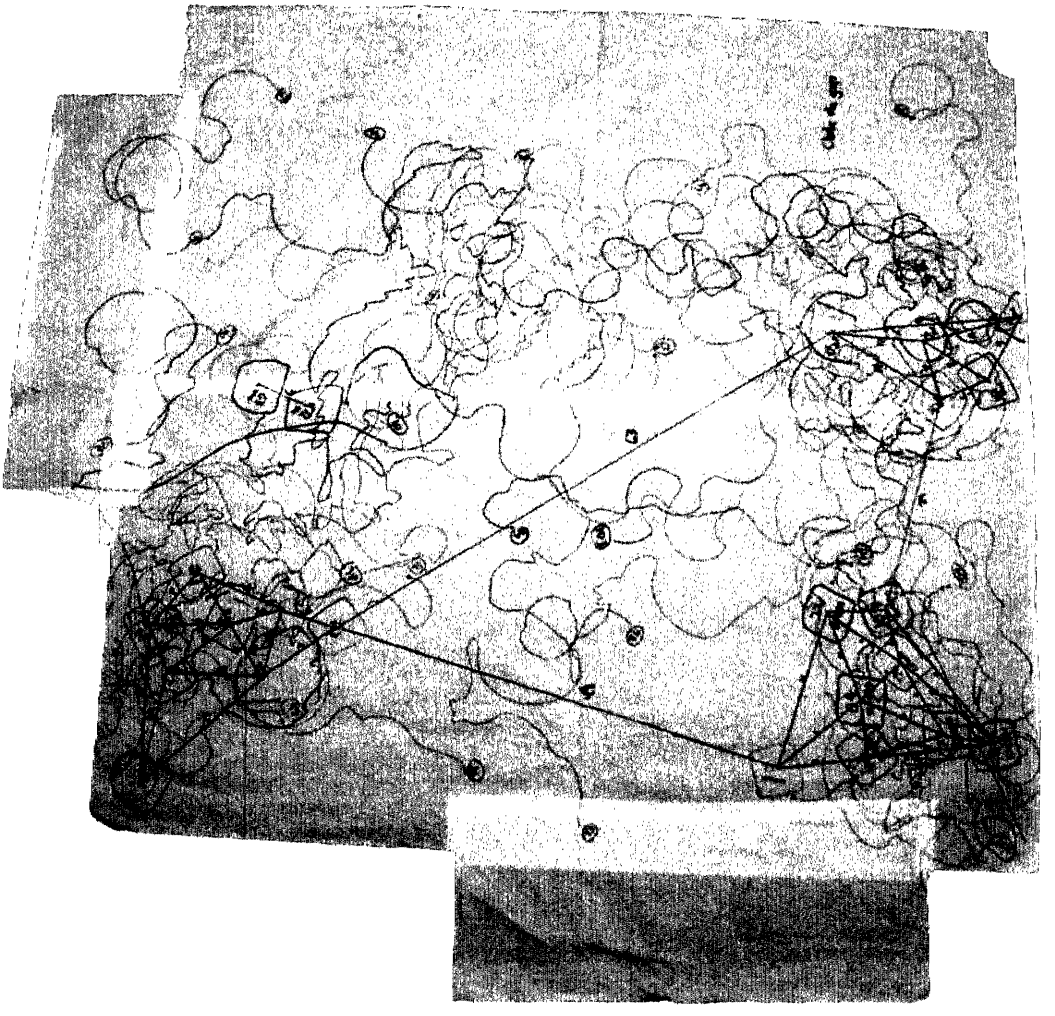
Notes:

clusterOnSource – convex hulls



Notes:

clusterIntoGroups



Notes

References

[1] iRobot Corporation. 63 South Ave., Burlington, MA 01803. www.irobot.com

[2] Express Logic Corporation, 11423 West Bernardo Court, San Diego, CA. 92127, www.expresslogic.com

[3] Cortes, J., S. Martinez, T. Karatas, and F. Bullo. Coverage control for mobile sensing networks. In Proceedings of the IEEE International Conference on Robotics and Automation, pages 1327--1332, Arlington, VA, May 2002.

[4] Arya, S. and A. Vigneron. "Approximating a Voronoi Cell". HKUST Theoretical Computer Science Center Research Report HKUST-TCSC-2003-10, Hong Kong University of Science and Techology, available at www.comp.nus.edu.sg/~antoine/avn.pdf, 2003.

[5] Brooks, R. "A robust layered control system for a mobile robot". In IEEE Journal of Robotics and Automation, RA-2, pp.14-23, 1986.

[6] Abramson, N. "The Aloha System - Another Alternative for Computer Communications". In Proc. Fall Joint Cornput. Conf., AFIPS Conf., page 37, 1970.

[7] Intanagonwiwat, C., R. Govindan and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In Proc. Sixth Annual International Conference on Mobile Computing and Networks, 2000.

[8] Payton, D., M. Daily, R. Estowski, M. Howard, and C. Lee. "Pheromone Robotics". In Autonomous Robots, vol. 11, pp.319-324, 2001.

[9] Weisstein, Eric. "Circle Packing." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/CirclePacking.html

[10] Holldobler, Bert, E. O. Wilson, "The Ants", The Belknap Press of Harvard University Press, Cambridge, Massachusetts, 1990

[11] Holldobler, Bert and Edward O. Wilson, "Journey to the Ants", The Belknap Press of Harvard University Press, Cambridge, Massachusetts, 1994

[12] Heinrich, Bernd, "Bumblebee Economics", Harvard University Press, Cambridge, Massachusetts, 1981

[13] Cormen, Thomas H., Charles E. Leiserson and Ronald L. Rivest, "Introduction to Algorithms", The MIT Press, Cambridge, Massachusetts, 1990.

[14] Abelson, H., D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight, R. Nagpal, E. Rauch, G. Sussman and R. Weiss. "Amorphous Computing". MIT AI Memo 1665, August 1999. http://www.swiss.ai.mit.edu/projects/amorphous/paperlisting.html

[15] Coore, D.. "Botanical Computing: A developmental Approach to Generating Interconnect Topologies on an Amorphous Computer". MIT Ph.D. Thesis. February 1999

[16] Braitenberg, Valentino, "Vehicles: Experiments in Synthetic Psychology", MIT Press, Cambridge, Massachusetts, 1984

[17] Balch , Tucker and Ronald C. Arkin. "Behavior-based Formation Control for Multi-robot Teams". IEEE Transactions on Robotics and Automation. 1999. http://www.cc.gatech.edu/aimosaic/robot-lab/mrl-online-publications.html

[18] Bonabeau, Eric, Andrej Sobkowski, Guy Theraulaz, Jean-Louis Deneubourg. [98-01-004]. "Adaptive Task Allocation Inspired by a Model of Division of Labor in Social Insects". Bio Computation and Emergent Computing, edited by D. Lundh, B. Olsson, and A. Narayanan, pp. 36--45, World Scientific, 1997. http://www.santafe.edu/sfi/publications/working-papers.html

[19] Bonabeau, Eric, Guy Theraulaz, Bertrand Schatz, and Jean-Louis Deneubourg [99-01-006] Response Threshold Model of Division of Labour in a Ponerine Ant. Santa Fe Institute. http://www.santafe.edu/sfi/publications/working-papers.html

[20] Botee , Hozefa M. and Eric Bonabeau [99-01-009]Evolving Ant Colony Optimization. Santa Fe Institute. http://www.santafe.edu/sfi/publications/working-papers.html

[21] Brooks, Rodney. "A Robust Layered Control System for a Mobile Robot". MIT Artificial Intelligence Lab. A.I. Memo 864. September 1985

[22] Brooks, Rodney. "A Robot that Walks; Emergent Behaviors from a carefully Evolved Neetwork". MIT Artificial Intelligence Lab. A.I. Memo 1091. September 1989

[23] W. Burgard, D. Fox, and S. Thrun. Active Mobile Robot Localization , Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97), Nagoya, Japan 1997. http://www.cs.bonn.edu/~wolfram/

[24] W. Burgard, D. Fox, M. Moors, R. Simmons, and S. Thrun. "Collaborative Multi-Robot Exploration". In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), San Francisco, CA, 2000. IEEE. http://citeseer.nj.nec.com/burgard00collaborative.html

[25] Chun, Li, Zhiqiang Zheng, and Wensen Chang. "A Decentralized Approach to the Conflict-free Motion Planning for Multiple Mobile Robots". Proceedings of the 1999 IEEE International Conference on Robotics and Automation. Detroit, Michigan. May 1999. pp.1544-1549.

[26] Delgado, Jordi and Ricard V. Sole. [98-08-069]. Self-Synchronization and Task Fulfilment in Social Insects. Santa Fe Institute. http://www.santafe.edu/sfi/publications/working-papers.html

[27] Desai, Jaydev P., Vijay Kumar, and James P. Ostrowski. "Control Changes in Formation for a Team of Mobile Robots." Proceedings of the 1999 IEEE International Conference on Robotics and Automation. Detroit, Michigan. May 1999. pp.1556-1561.

[28] Dorigo M., V. Maniezzo & A. Colorni (1996). The Ant System: Optimization by a Colony of Cooperating Agents. IEEE Transactions on Systems, Man, and Cybernetics-Part B, 26(1):29-41 http://iridia.ulb.ac.be/~mdorigo/ACO/ACO.html#Bib

[29] Dorigo M., G. Di Caro & L. M. Gambardella (1999). Ant Algorithms for Discrete Optimization. Artificial Life, 5(2):137-172. Also available as Tech.Rep.IRIDIA/98-10, Université Libre de Bruxelles, Belgium. ftp://iridia.ulb.ac.be/pub/mdorigo/journals/IJ.23-alife99.pdf

[30] Gage, D.W., "Many-Robot MCM Search Systems," Proc. Autonomous Vehicles in Mine Countermeasures Symposium, pp. 9.56-9.64, Monterey, CA, 4-7 April, 1995 http://www.nosc.mil/robots/pubs/PubsIdx.html

[31] Gage, D.W., "Development and Command-Control Tools for Many-Robot Systems," Proc. SPIE Microrobotics and Micromechanical Systems, Vol. 2593, pp. 121-129, Philadelphia, PA, 23-24 October, 1995. http://www.nosc.mil/robots/pubs/PubsIdx.html

[32] Gage, D.W., "How to Communicate with Zillions of Robots", Proceedings of SPIE Mobile Robots VIII, Boston MA, 9-10 September 1993, volume 2058, pp 250-257. http://www.nosc.mil/robots/pubs/PubsIdx.html

[33] Gage, D.W., "Randomized Search Strategies with Imperfect Sensors", Proceedings of SPIE Mobile Robots VIII, Boston MA, 9-10 September 1993, volume 2058, pp 270-279. http://www.nosc.mil/robots/pubs/PubsIdx.html.

[34] J-S. Gutmann, W. Burgard, D. Fox, and K. Konolige. An Experimental Comparison of Localization Methods. International Conference on Intelligent Robots and Systems (IROS 98), Victoria, Canada, October 1998. http://www.cs.bonn.edu/~wolfram/

[35] H. Hu, I.D. Kelly D.A. Keating and D. Vinagre (1998) Coordination of multiple mobile robots via communication. http://www.uwe.ac.uk/facults/eng/ias/iankelly/

[36] Jung, David and Alexander Zelinsky, Grounded Symbolic Communication Between Heterogenoeus Cooperating Robots, Autonomous Robots journal, special issue on Heterogeneous Multi-robot Systems, Kluwer Academic Publishers, Balch, Tucker and Parker, Lynne E. (eds.), vol 8, no 3, pp269-292, June 2000

[37] Jung, David and Alexander Zelinsky, "Integrating Spatial and Topological Navigation in a Behavior-Based Multi-Robot Application", International Conference on Intelligent Robots and Systems (IROS99), Kyongju, Korea, October, 1999.

[38] Jung, David and Alexander Zelinsky, An Architecture for Distributed Cooperative-Planning in a Behaviour-based Multi-robot System, Journal of Robotics and Autonomous Systems, 26: 149-174, 1999.

[39] Kube, C. Ronald, and Eric Bonabeau [99-01-008] Cooperative Transport By Ants and Robots. http://www.santafe.edu/sfi/publications/working-papers.html

[40] Maja J Mataric, "Interaction and Intelligent Behavior". A.I. Memo 1495. MIT Ph.D. Thesis. May 1994

[41] Maja J Mataric, Martin Nilsson and Kristian Simsarian, "Cooperative Multi-Robot Box-Pushing" Proceedings, IROS-95, Pittsburgh, PA, 1995, 556-561. . http://www-robotics.usc.edu/~maja/publications.html

[42] Maja J Mataric, "Using Communication to Reduce Locality in Distributed Multi-Agent Learning", Journal of Experimental and Theoretical Artificial Intelligence, special issue on Learning in DAI Systems, Gerhard Weiss, ed., 10(3), Jul-Sep, 1998, 357-369. http://www-robotics.usc.edu/~maja/publications.html

[43] McLurkin, James. "The Ants: A Community of Microrobots". Massachusetts Institute of Technology. Bachelor's Thesis. June 1995

[44] McLurkin, James. "Algorithms for Distributed Sensor Networks". University of California Berkeley. Master's Thesis. June 1999

[45] Nagpal, R. "Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics" MIT Ph.D. Thesis. June 2001

[46] Parker, Lynne. "Adaptive Heterogeneous Multi-Robot Teams". Neurocomputing, special issue of NEURAP '98: Neural Networks and Their Applications 1999, vol. 28, pp. 75-92.

[47] Russell, R. Andrew. "Laying and Sensing Odor Markings as a Strategy for assisting Mobile Robot Navigation Tasks" IEEE Robotics and Automation Magazine, Setpember 1995 pp.3-9

[48] Schneider-Fontan , Miguel and Maja J Mataric, "Territorial Multi-Robot Task Division", IEEE Transactions on Robotics and Automation, 14(5), Oct 1998 http://www-robotics.usc.edu/~maja/publications.html.

[49] Seeley, Thomas D. "The Wisdom of the Hive" Harvard University Press. Cambridge, Massachusetts. 1995

[50] Solé, Ricard V., Eric Bonabeau, Jordi Delgado, Pau Fernández and Jesus Marín [99-10-074] Pattern Formation and Optimization in Army Ant Raids. http://www.santafe.edu/sfi/publications/working-papers.html

[51] Werger , Barry Brian and Maja J Mataric, 'Robotic "Food' Chains: Externalization of State and Program for Minimal-Agent Foraging" in Proceedings, From Animals to Animats 4, Fourth International Conference on Simulation of Adaptive Behavior (SAB-96), Pattie Maes, Maja Mataric, Jean-Arcady Meyer, Jordan Pollack, and Stewart W. Wilson, eds, MIT Press/Bradford Books 1996, 625-634. http://www-robotics.usc.edu/pub_autonomous.html

[52] Yamauchi, Brian. "Frontier-Based Exploration Using Multiple Robots" Proceedings of Autonomous Agents 1998, Minneapolis, MN. pp 47-53.

[53] Gawlick, R, R. Segala, J. Soegaard-Andersen, and N. Lynch. Liveness in timed and untimed systems. Technical Report MIT/LCS/TR-587, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 02139, December 1993.