DREU Summer 2014 Project Report:
A Distributed Exploration Algorithm Implementation for Scribbler Robots

Student: Diana Ruggiero
Bard College
dr7745@bard.edu

Project Advisor: Prof. Maria Gini
University of Minnesota
gini@cs.umn.edu

Introduction:

In a disaster scenario, a team of robots could be used to explore a damaged or potentially dangerous building in order to provide maps of the area before human rescuers enter. The benefit of using a distributed algorithm is that individual robots can be damaged or lost without compromising the algorithm as a whole. My DREU summer project was to implement a distributed exploration algorithm on Scribbler robots. This raised a number of challenges, mostly regarding the specific hardware limitations of the Scribblers. Although some of these hardware limitations could be overcome with software solutions, ultimately our wall-following based implementation failed because the obstacle sensor on which the basic navigation was based was not as robust as first assumed.

Past Work:

Previous work in the area of distributed exploration algorithms has focused on both communication protocols as well as exploration/navigational logic. Ludwig and Gini (2006)[1] explicated the use of wireless signal intensity as a means of determining robot proximity instead of relying on sensors to recognize nearby neighbors. It was also demonstrated that dispersive coverage can be achieved without the robots having explicit knowledge of relative neighbor locations, as proximity information alone is adequate.

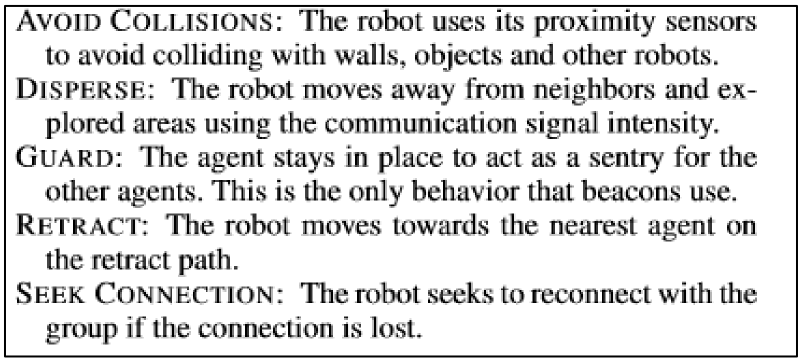
Damer et. al (2006)[2] proposed two different distributed algorithms for small robots: Clique-Intensity, which uses a connectivity graph of dispersed robots weighted by signal strength in order to triangulate the robot formation for maximal area coverage, and Backbone Dispersion, which maintains a stable connected group of stationary robots as the “backbone” and allows other robots to explore freely as long as a robot is either in range of the backbone or in range of another robot in range of the backbone. The backbone can be expanded with more robots to increase its reach.

Kurazume and Hirose (2000)[3] describe an algorithm known as Cooperative Positioning System, which assigns each robot to one of two groups. One group explores freely while the other group remains stationary as a landmark for the exploring group. The two groups switch roles in a leapfrogging fashion in order to explore the area.

Jensen and Gini (2013)[4] propose the Rolling Dispersion Algorithm, which is very similar to the algorithm on which our Scribbler implementation was based. In the Rolling Dispersion Algorithm, robots explore as far as they can while staying in signal range of the group of robots and drop beacons where they have explored in order to mark rooms and leave a path back to the exit of the area. The algorithm uses a system of sentry and explorer robots which allow the group to move to explore the area. Robots can send requests for other robots to explore an area beyond the group range in an unexplored direction, at which point an available robot will respond and push forward.

Algorithm Description:

We attempted to implement an algorithm that was based off the Rolling Dispersion Algorithm known as the Sweep Exploration Algorithm (SEA), described in Jensen et. al (2014)[5]. Our ultimate implementation was quite different from the original SEA due to simplifications required by the Scribbler hardware. SEA works by having the group of robots disperse and move into the branches of the area, having the group as a whole explore one branch in full before retracted out of the branch and moving on to the next unexplored path in a manner quite similar to a depth-first search. Areas that have been explored are marked by beacons, which serve to both prevent repeated exploration of dead ends/explored areas as well as providing a pathway back to the entrance for the robots to follow once they have finished (this is how branch retraction is handled in general). By dropping beacons behind them, robots towards the back of the group can be called up to the frontier of the currently explored branch. Proximity is determined by wireless signal range and ideally all robots remain in range of at least one other robot so the group moves together. Figure 1 shows the 5 different behaviors involved in the SEA, which were adapted and condensed for our later implementation.



AVOID COLLISIONS: The robot uses its proximity sensors to avoid colliding with walls, objects and other robots.
DISPERSE: The robot moves away from neighbors and explored areas using the communication signal intensity.
GUARD: The agent stays in place to act as a sentry for the other agents. This is the only behavior that beacons use.
RETRACT: The robot moves towards the nearest agent on the retract path.
SEEK CONNECTION: The robot seeks to reconnect with the group if the connection is lost.

Figure 1: Behaviors of SEA [Jensen et. al (2014)]

SEA attempts to minimize communication load, so there are 7 different state messages that can be sent between robots and beacons in order to coordinate the desired behaviors, as seen in Figure 2:

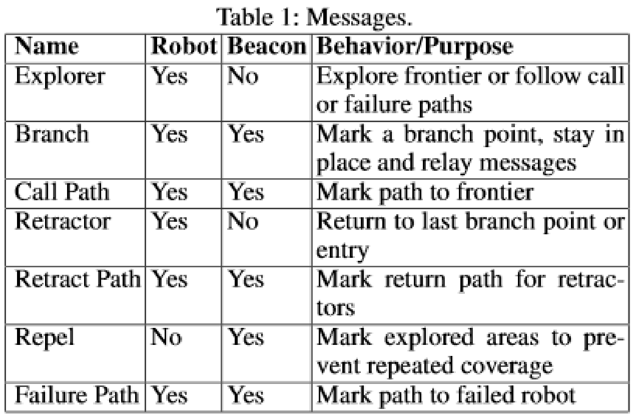


Table 1: Messages.

Name	Robot	Beacon	Behavior/Purpose
Explorer	Yes	No	Explore frontier or follow call or failure paths
Branch	Yes	Yes	Mark a branch point, stay in place and relay messages
Call Path	Yes	Yes	Mark path to frontier
Retractor	Yes	No	Return to last branch point or entry
Retract Path	Yes	Yes	Mark return path for retractors
Repel	No	Yes	Mark explored areas to prevent repeated coverage
Failure Path	Yes	Yes	Mark path to failed robot

Figure 2: State Messages of SEA [Jensen et. al (2014)]

Implementation Design:

The much-simplified and greatly modified attempt to implement this algorithm defined 3 behaviors and 4 different states (as opposed to SEA's 5 behaviors and 7 states). In the main loop of each robot, it is the states—in addition to environmental information—that determine what behaviors will be executed. Charts describing these behaviors and states are found below:

Behavior	**Description**
FollowWall	Uses obstacle sensor to follow right-hand wall until an opening is discovered.
Guard	Do nothing, remain in place.
ExploreRoom	After finding an opening in the wall and the numbered intersection is marked as a room, "explore" it.

State	**Description**
Offline	As the starting state, waits until it is needed as explorer. As ending state, sits and waits for other robots to finish.
Explorer	Follows wall until it comes upon opening at which point it should call for another robot to explore further.
Idle	The state for when a robot is no longer offline but is waiting for an incoming request. Robot will enter Idle state after calling for an explorer.
Responding	Has received request to be explorer and decided that it is the closest available robot, follows wall to the intersection to which it has been called and explores beyond the robot that made the request.

In simple terms, the implementation works by having a single explorer robot follow the wall until it comes upon an opening, at which point it reads a nearby sign to get information about that intersection or room. If it is explored, the explorer ignores it and continues along the wall. If it is unexplored, the explorer stops and calls for the closest available robot to explore beyond it, as being "in-range" is defined as being only one intersection away from another robot in the group. Robots that are called to explore further will follow the wall until they arrive at the intersection they must explore. This way, all branches and rooms will be explored in a depth-search-first fashion until every room/intersection is marked as explored. The path-retraction as present in the SEA isn’t present in our implementation, as the robots simply follow the wall in only one direction. Retracting out of a fully explored path just means following the wall all the way around until the exit of the branch is detected.

Because the SEA algorithm involves the navigation of and communication between a number of agents, our implementation was executed using an object-oriented paradigm in which each robot was represented as an object of the RobotObject class. The entire protocol is run by the Controller.py program, which defines the intersection and room layout and then connects to all the robots for which is has been given the COM port number. Each robot begins in the OFFLINE state and their locations are all set to be at the starting intersection. The Controller then sets the first robot in the robot list to the state EXPLORER and then loops through each robot, executing their main function as long as not all of the robots have returned to start, at which point all of their states would again be OFFLINE and the algorithm would terminate.

Although the original algorithm makes heavy use of beacons, because of the lack of available hardware, point-to-point communication, and any way for the Scribbler to “drop” a beacon we instead implemented each intersection as an IntersectionObject which could be set to be “explored.” This way, when a robot arrives at an intersection and reads the sign it will be able to query the intersection as to whether or not it is explored—if it is, the robot will keep going and if not it will stop and call another robot to explore it. These IntersectionObjects were also what were used to build the pre-stored area information about the area, as each IntersectionObject keeps a list of other intersections in range of themselves. When a robot signals the computer that it has arrived at a certain numbered intersection by reading the encoded sign, the robot's location is updated to be at that intersection. Thus, any robots located at other intersections that are set to be in range of that robot's present location are considered to be neighbors to that robot and can receive requests and simulated point-to-point messages from it.

The sign encoding was done with a 3-bit encoding system on pink paper cylinders each with 3 horizontal stripes that could be either black or white. This makes it easy for robots to take a picture, detect the pink sign, and then read the stripes like a barcode to figure out what number intersection or room they were at in order to find out whether or not it had been explored. Robots take pictures after detecting room openings and after they are done exploring a room in order to find out where they are in relation to explored/unexplored areas.

The algorithm terminates once every robot is in the OFFLINE state, as this is the state the robot enters once it has passed the final sign (the first robot to reach the final sign will set that sign to "explored"). If every robot has done so, it means that all of the other explorable area signs have been reached and explored, as the robots will not move on as a group until they have explored all of the areas around them.

Hardware Limitations:

Ultimately, implementing this algorithm (or similar distributed exploration algorithms) on the Scribblers requires a number of compensatory modifications that move functions that would otherwise be delegated to specific hardware over to software simulations that use the hardware that is available.

Communication:

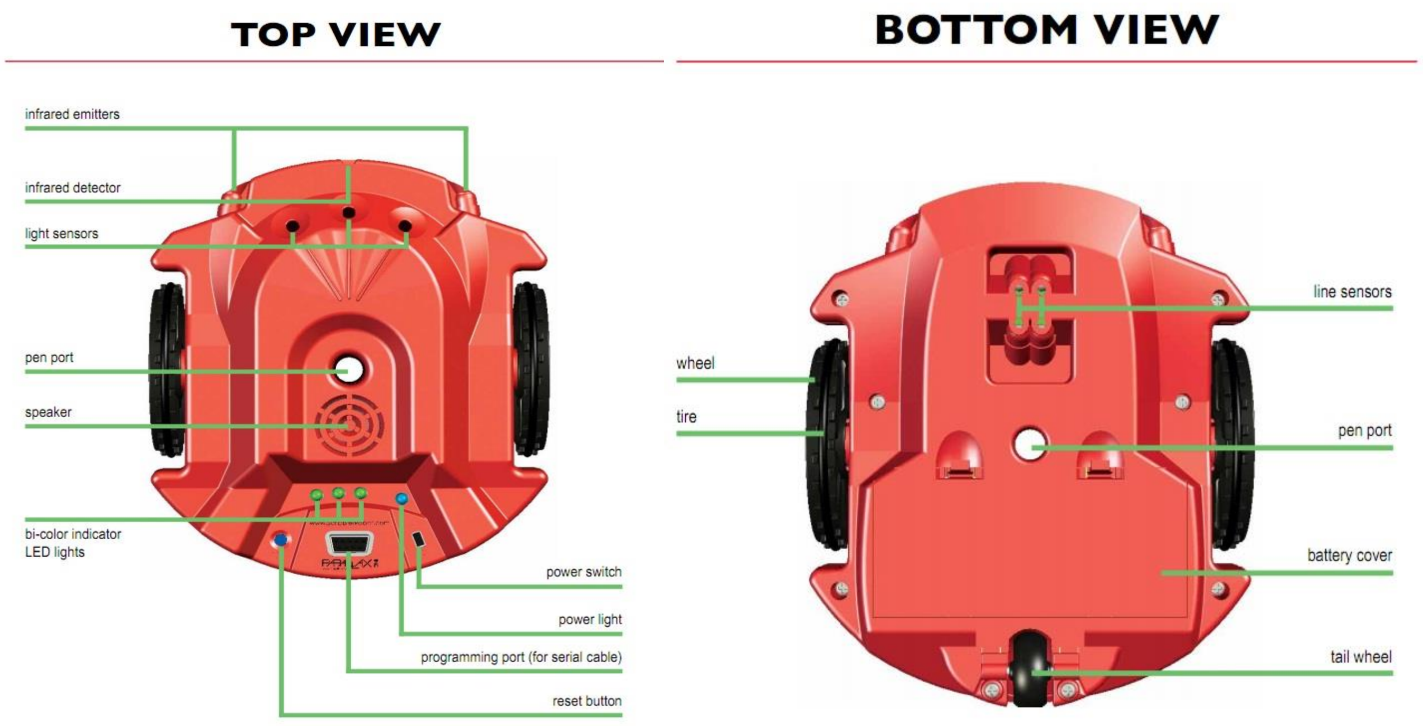
The lack of point-to-point communication ability means that communications amongst the robots must be simulated by what ends up being a highly centralized system. Furthermore, because of this limitation the functions of the dropped beacons in the original algorithm is instead built into the centralized system as well, with areas being marked as explored or unexplored in a prewritten central database. Proximity, instead of being determined by the robots within the environment through signaling, was handled entirely by the central system comparing the robots' reported locations to the pre-stored information about the layout of the area.

The simulation of point-to-point communication is handled by the RobotObject class's function addIncomingRequest( ), which can be accessed by other robots when they need to call another robot to an intersection to continue exploration. When a robot needs backup, it will loop through its own stored list of neighboring robots and add requests to these robots. This list of neighbors is updated each time the robot reaches a new intersection by using pre-stored information about the area's layout to find out how far away the other robots are and which ones can receive a request for help. addIncomingRequest( ) takes an IntersectionObject as a parameter, which tells the requested helper where they are needed to continue exploration. The closest available neighbor robot should be the one to respond. Although addIncomingRequest( ) and the neighbor-calling process is distributed amongst objects in the code, when run on the Scribblers every communication request is all done on the central controlling computer—it is never the robots actually communicating with each other, but rather their object representations on the central computer using location information (from the intersection signs) as reported by the actual robots in the area. As mentioned before, this approach also requires the central computer to have accessible information about the entire area that is to be explored so the robot objects can know who their neighbors are without using something like signal intensity to figure out proximity.

Sensing and Off-board Navigation

Second, the robots' vision and ability to quickly process images was made slower because of the constant need to do offboard image processing—meaning that the robots' navigational capacities are limited by the speed of the Bluetooth connection. This is also why the initial line-following attempts were failures, as it affects every sensing capability on the robot. Because of this, the cameras were used only to read signs in the environment that then allowed the robot object to query the central controller for pre-stored information about the area.

The Scribbler's camera is useful to examine the environment, and our implementation uses it in order to read colored signs around the area. Each robot object has an instance of the class SignReaderObject, which takes care of all of the sign-reading image processing functions. However, the usefulness of the camera is impacted by the speed at which images can be



Figures 2 & 3: Scribbler robot diagram with sensors (Fluke not attached). [pololu.com]

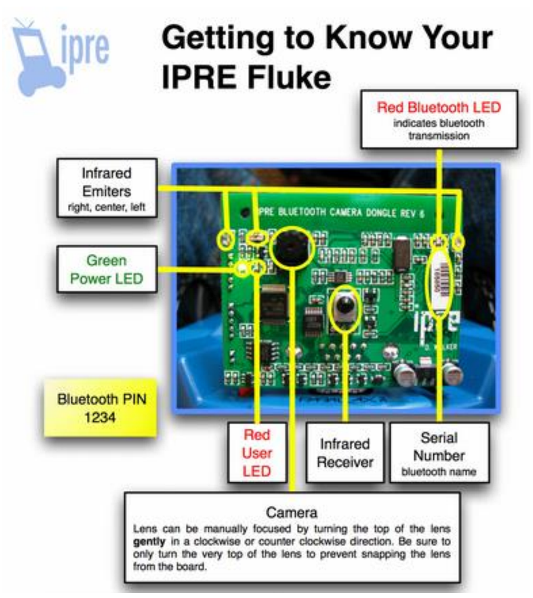
processed. Processing the image on the central computer can take some time, although this time can be minimized by how the image processing is coded. What also takes time is for the image to travel over Bluetooth from the Scribbler to the computer, and then for the Scribbler to receive the computer's navigational instruction based on the image it took.

This process is what happens with every sensor. The line-sensor, obstacle sensor, light sensor, and others all take readings from the environment and then send the results over Bluetooth to the computer. It is up to the computer to use the sensor results to decide on a next action for the connected robot, which it then sends back over Bluetooth. This causes a communications lag that made the line-sensor unusable (due to its requirement of high precision) and other functions slower and less accurate. Although the Scribblers come with an on-board, firmware installed line-following function that does not need to send the sensor results over Bluetooth, updating the robot's firmware to use with Calico/Myro gets rid of this function. Our attempts at re-instating this capability did not work—hence why we attempted to use the obstacle sensor instead, as it required less precision to navigate successfully and was not as affected by the Bluetooth lag.

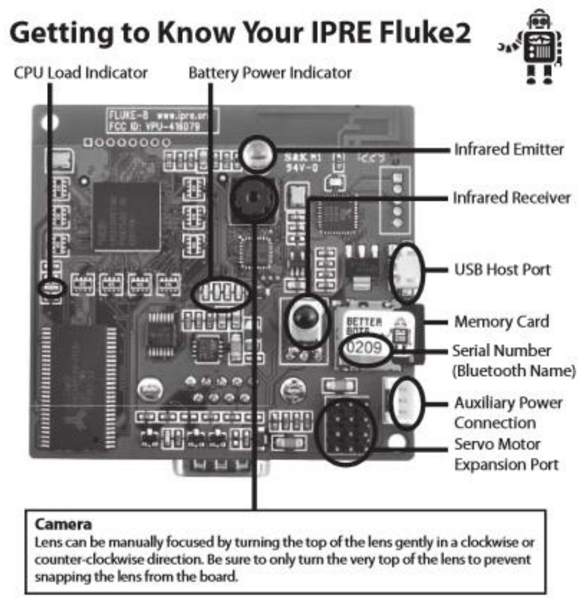
In essence, having all of the Scribbler's instructions located outside of the robot itself means that each robot in our implementation really exists in two parts—the object representation in the central computer, and the physical robot that does what the off-board portion tells it to. Having a communications lag within each robot's individual functioning is not ideal and seriously impacts the navigational capabilities of the Scribblers.

Obstacle Sensor and Unreliable Wall-Following:

Third, our approach had us using the obstacle sensors of the robot in order to move through the area and detect room openings, which on the newer Fluke models lack lateral sensing capabilities. This problem was initially unknown because although looking at the physical Fluke2 hardware one sees only one infrared emitter, there is no explicit indication (in documentation or otherwise, as far as we know) that this piece only points forward. It is especially confusing because the obstacle sensor still returns a list of three often wildly different



Lens can be manually focused by turning the top of the lens **gently** in a clockwise or counter clockwise direction. Be sure to only turn the very top of the lens to prevent snapping the lens from the board.


Getting to Know Your IPRE Fluke2
CPU Load Indicator
Battery Power Indicator
Infrared Emitter
Infrared Receiver
USB Host Port
Memory Card
Serial Number (Bluetooth Name)
Auxiliary Power Connection
Servo Motor Expansion Port
Camera
Lens can be manually focused by turning the top of the lens gently in a clockwise or counter-clockwise direction. Be sure to only turn the very top of the lens to prevent snapping the lens from the board.

Figures 4 & 5: On the left, the older Fluke model with right, center, and left infrared emitters. On the right, the newer Fluke model with one central infrared emitter. [betterbots.com]

values, giving the impression that it is still acting like the first Fluke and is returning a Left, Center, and Right infrared obstacle-distance value. However, it turns out that these three values are all coming from the same sensor although it is unknown exactly why there are three different readings. The Scribbler/Fluke documentation available online does not seem to have made this change explicit, but after speaking to one of the developers they confirmed that there is indeed just the one forward sensor on the newer models.

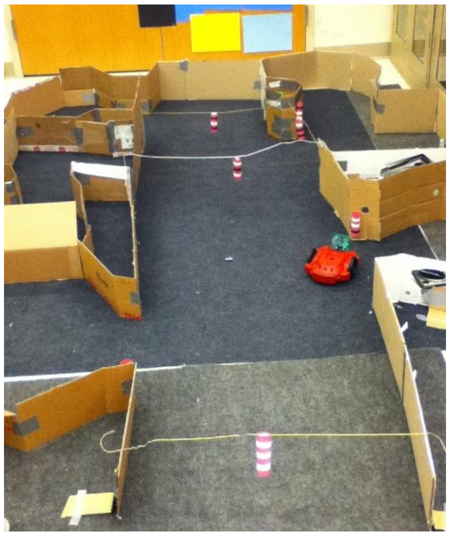


Figure 6: Final test area setup

Although the robots would still sometimes find the desired openings (in some test setups finding the opening almost every single time), having a forward-pointing sensor looking for wall gaps to the right-hand side of the robot was not sufficiently accurate when we tried using it in our final test area. While the first two hardware limitations can be compensated for with software solutions (although they *do* in some sense undermine the decentralized and exploratory purpose of the algorithm), this final problem cannot really be circumvented satisfactorily.

Conclusions:

The description of the final implementation makes it clear that many modifications/simplifications were made to the algorithm in order to try and make it work on the Scribbler hardware. Many hardware limitations had to be dealt with by using software simulations to augment the robots' capabilities. The first category of limitation was robot communication—the lack of point-to-point communication meant we had to simulate point-to-point communication within the central computer. The second category was in the robots' delayed sensing capabilities caused in part by the lag in the Bluetooth communication, impacting the robots' navigation. The third category is which the infrared obstacle sensor specifically, and the change made on the newer Flukes that took away lateral sensing and made it so our wall-following protocol was unreliable in detecting rooms and intersections. The first two categories of limitations could be augmented by software, but the third could not be reliably overcome.

Hence, it would appear that if Scribblers are to be used for an exploration algorithm one ought not rely on the newer model's obstacle sensors, at least not in the context of wall-following and room entrance detection. If the line-following abilities can be implemented in the robots' firmware successfully, this approach (alongside some of the aforementioned software modifications) might be more successful in building a working demo. However, perhaps there is an alternative small and cheap robot model more suited to this task.

References:

[1] Luke Ludwig and Maria Gini. Robotic Swarm Dispersion Using Wireless Intensity Signals. In *Proc. Int'l Symp. on Distributed Autonomous Robotic Systems*, pp. 135–144, July 2006.

[2] Stephen Damer, Luke Ludwig, Monica Anderson LaPoint, Maria Gini, and Nikolaos Papanikolopoulos. Dispersion and exploration algorithms for robots in unknown environments. In *SPIE*, April 2006.

[3] Ryo Kurazume and Shigeo Hirose. An Experimental Study of a Cooperative Positioning System. In *Autonomous Robots*, v.8 n.1, pp.43-52, January 2000.

[4] Elizabeth Jensen and Maria Gini. Rolling Dispersion for Robot Teams. In *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 2473–2479, 2013.

[5] Elizabeth Jensen, Ernesto Nunes, and Maria Gini, Communication Restricted Exploration for Robot Teams. In *Proc. Multiagent Interaction without Prior Coordination.* July 2014.