2010

Control of self-reconfigurable robot teams for sensor placement

Jacob Hays

Control of Self-Reconfigurable Robot Teams for Sensor Placement

by

Jacob Hays, B.S.

THESIS

Presented to the Faculty of the Golisano College of Computer and Information Sciences
Rochester Institute of Technology
in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

Rochester Institute of Technology

May 2010

Control of Self-Reconfigurable Robot Teams for Sensor Placement

APPROVED BY

SUPERVISING COMMITTEE:

Dr. Zack Butler, Supervisor

Dr. Richard Zanibbi, Reader

Prof. Alan Kaminsky, Observer

Abstract

Control of Self-Reconfigurable Robot Teams for Sensor Placement

Jacob Hays, M.S.

Rochester Institute of Technology, 2010

Supervisor: Dr. Zack Butler

Self Reconfigurable Robots (SRRs) are a system of many simple modules that can rearrange themselves to work together and better perform complicated tasks. They are in theory more extensible then traditional robotics. We investigate the particular problem of using SRRs to both explore and survey an unknown environment. The environment is explored by using the Robots internal sensors, and surveyed by placing a limited number of static sensors at ideal locations. The advantage of SRRs is that they can adapt to terrain difficulty by adjusting the number of individual robots on the field by reorganizing its modules. We test a distributed task driven implementation based on the ALLIANCE architecture in a simulated environment. The results show that SRRs are both able to cooperatively explore the environment as well as place sensors in useful locations, getting good results.

Table of Contents

List of Tables

List of Figures

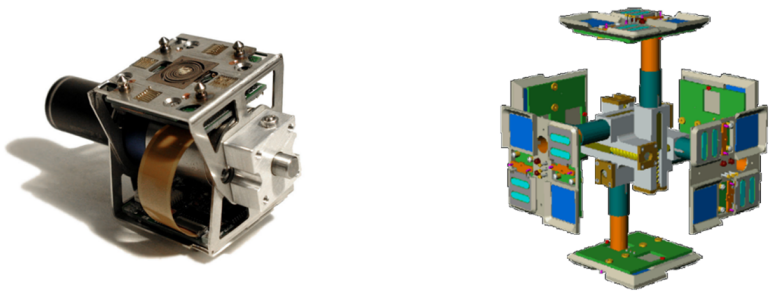
Chapter 1

Introduction

Self-Reconfiguring Robots (SRR) are a system of many simple modules which can rearrange themselves to perform various tasks. While a single complex robot may be able to do a particular task well, a system of many reconfigurable modules could do that task, as well as others by virtue of rearranging itself. Along with versatility, it also adds the benefit of higher fault tolerance, as bad modules can be replaced easily. While SRRs have great potential, there are many unsolved problems in the field of self-reconfiguring robots, and they are far from competing with traditional robots currently. This include issues with creating working physical implementations, as well as algorithmic problems.

There are two main types of physical implementations of these robots, lattice and chain. In Lattice SRR systems, the modules are shaped so they can be densely packed into a space filled solid. Telecubes are an example of this type [18]. Each Telecube module can connect to another module on each of its 6 sides. The sides can also expand outwards over to over half the compact cubes width. Each cube has its own processor and power source, and packed together can form different 3D structures. The expansion and contraction of each sides along with changing connections can let a group of modules change from one shape to others.

An example implementation of a chain type SRR is the Polybot [21]. The Polybot consists of two types of modules, a Link and a Node. A Link module contains 2 connection plates, with a single motorized axis of rotation between. A Node module is simply a cube with 6 connection ports, extra power supply, and no means of movement. A full robot is a chain of link modules, with nodes to provide connections. A series of these links is able to move around an environment in a snake, a wheel, and a spider walker configuration. A system



(a) Polybot (Chain) (b) Telecube (Lattice)

Figure 1.1: Physical implementations of SRRs.

of these can switch between the different configurations. The creators thought that a system such as theirs could be useful in space applications, due to the lack of dirt and gravity. Chains of these modules could traverse a surface of a spacecraft, forming fewer longer or greater shorter chains depending on the task. Some systems are a hybrid of both types, such as M-TRAN (Modular Transformer) [10]. It has both the ability to form a space filling solid, as well as form a chain of modules.

The current physical implementations of SRRs are far from what we believe they could do. Many limitations now are based on the size, power, cost, and reliability of the modules. The smaller modules are, the more adaptable the entire system can become, but this causes the complexity of each module to rise. For SRR's to compete with traditional robotics, the modules need to be smaller, cheaper, and more power efficient then they currently are. One ultimate goal is to have them work at a nano-technology scale [15].

While many of the physical challenges in the field of SRR's are dependent on future advances in technology, many of the algorithmic challenges can be tackled now. Although the physical implementations are not perfected yet, swarms of thousands of modular robots can be simulated with today's systems. These algorithmic challenges generally deal with the control and planning of these modules. The system as a whole can change from one configuration to another, and every module has some part to play in the locomotion of the system. Unlike a traditional robot, with a centralized processor controlling everything, each module has to be able to function on its own, as well as func-

tion as part of the collective. Many different approaches have been put forth, some leaning more towards centralized planning, and others towards more distributed. Centralized control is easier to develop, but may not be scalable to having thousands or millions of modules in the system. Distributed control is harder to do correctly, but would hopefully be able to handle more modules. Another algorithmic challenge is how communication works within the system. This includes ad-hoc communication among individual modules as well as communication between groups of these modules. All performed tasks should take advantage of the inherent fault tolerance of SRRs. To be able to swap out modules as they go bad, and to react to unusual situations through reconfiguration.

1.1 Problem Statement

Self reconfiguring robot systems can be applied to many problems, and the purpose of this paper is to investigate a specific one. We explore how a group of self organized robots can explore an unknown and potentially difficult environment, and deploy a limited number of carried surveillance sensors within it to maximize the area surveyed. A possible scenario could have an SRR dropped into an unknown environment, where its goal is to place a series of acoustic sensors to detect audible movement of humans or vehicles. This is a complex task that consists of multiple steps and sub-goals. Because the environment is initially unknown, a significant part must be explored first if it wants to place sensors in anything resembling a "good" position. This problem was chosen because it should take advantage of the strengths of SRRs. A SRR can choose to split into multiple robots to explore parts of the environment, but then merge back together to deploy a sensor in an location which is difficult to get to. We look at how it can succeed at its main goal, to survey the most terrain, relative to the time taken and energy used to do so.

To investigate solutions to the problem, a simulation system to model these robots was built. Individual modules of the robots are not modeled, only groups of them, which are referred to as a blobs. Blobs can traverse the terrain and deploy the survey sensors it contains at points in the map. It's ability to move and explore is based on the modules it contains, and unique to

SRRs, it can split into multiple smaller and less capable blobs, to seek multiple goals at once. These smaller blobs can then merge back together if needed to seek difficult goals. The SRR system starts with a limited number of sensors and the simulation ends when all the sensors have been placed. The scope of the simulation is limited, so assumptions are made to simplify some aspects. Perfect wireless communication is assumed between the blobs. Other assumptions are made in issues of power management and robot reconfiguration. The focus is on investigating the cooperation between the blobs. How can robots with modular capabilities best cooperate to complete ordered complex tasks? A task based control system based on the ALLIANCE architecture was used to control the blobs in the simulations. For the experiments, attributes which test the blob's willingness to cooperate and when to advance to later tasks are adjusted. Results such as how much terrain was surveyed, how long the simulation took, and how much energy was used help draw conclusions on the best methods.

Chapter 2

Previous Work

The problem studied is a combination of many other smaller problems. Each area has been covered extensively in previous work, but there is little which combines them all. Some of the areas studied include multi-robot explore, ad-hoc sensor networks, multi-robot cooperation, approximations for surveillance problems, and SRR movement.

The issue of single and multi-robot exploration is one of the most widely studied. While most of the techniques do not directly mention reconfigurable teams of robots, most can be used in at least some capacity. This is because SRR systems are a subset of multi-robot systems. One method used often is known as frontier based exploration. In it, the environment is represented as a matrix of cells of known and unknown territory. The cells on the border of known territory are known as frontier cells, and are all possible next exploration destinations. When one robot decides its destination, the utility value of all nearby frontier cells is lowered for the other robots. When the next blob decides its next exploration destination, it takes into account both the utility of and distance to a frontier cell. This makes sure that different blobs do not overlap areas when exploring. This method is described in Yamauchi's work [20]. In it, real robots use laser sonar to create evidence grids for obstacles. They share maps amongst themselves, as well as their planned destinations. This allows them to cooperate with each other and explore an office environment. A team from Germany and the US also used this method on physical robots. [1] Their experiments showed that a team of robots coordinated this way took significantly less time to explore a fixed area then the same robots without coordination.

Other methods have been studied to decrease overall robot displacement during exploration, rather than the time taken [13]. These simulations

are run in a world of 2D polygons with holes. Robots can move about the world detecting both obstacles and other robots. In the implementation, robots work in pairs, with only one moving at a time. They switch back and forth the role of exploring an area, while keeping in range of each other. They end up exploring the environment in vertical stripes to conserve energy. The overall robot displacement is reduced compared to when they do not cooperate.

Research has also gone into developing general methods for cooperation of multi-agent systems. These systems are both distributed or centralized. One in particular is the ALLIANCE architecture [12]. It is a distributed fault tolerant method for control of multiple mobile robots. Many techniques from it are used in the current implementation, so it is discussed in detail in Section 3.2.

The surveillance of an area has been studied in many ways. The most famous of them is a classical problem, known as the Art Gallery problem. In it, you try to find the minimum number of guards placed in an room at a museum, so that all surfaces are within line of sight by at least one guard. The problem is NP-complete, so practical solutions of this surveillance problem are approximations instead of the optimal. Methods focus both on the traditional problem, as well as using electronic sensors to survey real terrain.

The world being surveyed can be represented in many different ways. The classical Art Gallery problem represents it as a 2D polygon with holes. González and Latombe of Stanford University approach this by using random sampling of points within the polygon to convert it into a set-cover problem [6]. Many view sheds at random points within the polygon are calculated, and at each, the areas of the polygon that are visible are tracked. The goal is to find the minimum number of points needed where all areas are surveyed. A greedy solution is tested, and found that it does produce a reasonable number of guards, but is not too close to optimal in many cases. To show that a random selection method is viable, they use the Vapnik-Čhervonenkis dimension of the sets to show that there is a high probability that results are reasonably close to the optimal solution.

A world can also be represented as a grid, with each cell being the probability it is surveyed by a sensor. In algorithms presented in [5], for each cell p_{ij} in the NxN matrix of probabilities, a view shed map sized NxN is

calculated which consists of the probability that each point is surveyed by a sensor being placed at p_{ij}. A sensor detection matrix is created sized N^4. A procedure selects the best sensor location each loop, based on how much is covered or missed at that location. Several case studies are run on 64, 100, or 400 point grids, with varying numbers of obstacles. They show that their sensor placement methods are much better then random and uniform placements.

Research has been done in the area of mobile self-deploying sensor networks. In these, not only do sensors have to find good locations to survey, but they also have to traverse the terrain to get to those locations. In research done in the Robotics Research Laboratories of the University of Southern California, potential fields are used to create a scalable distributed method for deploying mobile sensors in an unknown environment [8]. Sensor nodes have limited knowledge up to a certain distance of other nodes and obstacles, which they use to move independently of each other. To ensure a node spreads out from it's initial clustered position, each node and obstacle exerts a virtual force on it. Nodes are repelled away from each other and obstacles, and nodes will travel in the direction of the summed forces. In their simulated experiments, nodes spread rapidly away from each other and mostly filled the environment. When they reached equilibrium, nodes had quite even spacing, with no breaks in the coverage.

Mobile sensor networks are not just in the realm of simulation. Systems have been built for just that purpose. One specific system has two types of robots working as a team [17]. A *Scout* robot is a small cylindrical robot that can both roll around on terrain, and jump with a spring powered foot. It holds different sensors and cameras and is the eyes of the team. The *Ranger* is a much larger wheeled robot, which carries a Scout launcher. They have cameras of their own and can get information from the sensors of the Scouts. The Ranger can launch a Scout robot over obstacles and even through windows. Control of all the Scouts is centralized in the Ranger. The Scouts can navigate towards goals, avoid obstacles, and detect motion. In the experiments, the scouts would be deployed in cluttered rooms, and find a dark area to hide in. They would then observe the lighter areas of the room for motion.

Some recent research by Zack Butler and Eric Fabricant examined the

exploration problem dealing with Self-Reconfigurable Robots [2]. In this case, the number of robots on the field at any point is not constant, each one being a blob of modules. The teams of robots can rearrange themselves into different shapes, and can also split and merge to form an assortment of individual blobs of various sizes. They study how to most effectively get these reconfigurable teams to visit specified goal locations in an unknown environment by using a simulation

In the experiments, a variety of environments were explored, with goals randomly scattered around them. The movement of a blob is limited by the number of modules present within. It is not meant to model a specific SRR system, but instead is limited by basic rules. The larger the blob is, the steeper the slopes it's able to climb. Blobs have a view of their own local area, but store no map information or shared it amongst each other. The blobs use a reactive based approach to traverse the maps towards their current goal. Each blob is has a state machine based Navigation Control Unit, which keeps track of the current goals yet to be reached, and the states of all other blobs.

Several different NCU behaviors were experimented with in the simulations. They were called Generous, Greedy and Neutral. They affect how a blob ranks helping another blob vs. pursuing its own goals. A Limited behavior, where all cooperation is disabled is also tested. The results of the experiment showed that allowing reconfiguration of teams is beneficial compared to static teams of robots. The Generous, Neutral and Greedy behaviors all outperformed the Limited behavior in most of the tests. It showed that some parallelism is better than none, but it was not conclusive that one behavior was better than the others.

Chapter 3

Methodology and Simulation

This work is distinct in its approach and scope compared to previous work. Most previous work deals with solving part of this larger problem, whether that be multi agent exploration, optimal sensor placement, robot team cooperation, or adjustable SRR team sizes. All these separately studied problems are brought together in a single simulation, borrowing heavily from their approaches. The simulation environment and how the SRR robots are modeled is discussed first. The algorithms which control the simulated blobs are discussed afterwards, as much of them are dependent on knowing how the environment is represented.

3.1 Simulation Environment

The world this simulation takes place in is represented as a height map, where each 1x1 square is the same size as a single basic module of a self reconfigurable robot. These height maps are created from grayscale images, which can be scaled from their original value. This gives a high resolution terrain, but it is limited by the size of the image. Slope maps are created from the height map, which can help to determine traversability of an area. Visibility around the map is determined by which heights obscure others from sight.

The simulation does not represent each individual module of the robot. Instead, it represents each grouping of modules as a "Blob", and its capabilities are based off the Blob's contents. Its contents can be any number of generic modules, and any number of special type modules, which in this case, are the surveillance sensors it is trying to place around the map. The generic modules by themselves have capabilities for movement, power, vision, and communica-

Move:	Adjust position one cell in one of 8 cardinal directions.
Split:	Become two separate blobs on same cell.
Merge:	Merge with another blob, becoming one.
Deploy:	Deploy a contained sensor module at its location.
Pick up:	Pick up a deployed sensor module at its location.
Idle:	Do nothing.

Table 3.1: Low level actions

tion. The special modules could have their own effect on the capabilities, but in the case of the surveillance sensor, it acts like dead weight as it being carried around. Only the special modules can be deployed into the environment, where they have an effect separate from the blob. They can be picked up later by other Blobs, or stay deployed the rest of the entire simulation.

Each blob has its own knowledge of the environment and stores the status of itself and other blobs. They have to discover information about the map for themselves. Its attributes, such as what it can see, how it moves, and how it communicates with other blobs are determined by the contained modules. In the simulation, the sensors, mobility, and communication of a Blob are all modeled separately, in a way where they can be replaced easily with different versions. This way, any specific SRR system can be modeled in the simulation by replacing these sensor, movement, and communication models with ones specifically written for that SRR system. For this case, models were not developed for a particular SRR system. Instead, approximations for the capabilities of a lattice based SRR system were used.

Each blob is a separate thread in the simulation, which helps when running on multi-core machines. The simulation proceeds in fixed steps, where a Blob has the option to do one of the actions in Table 3.1. They are simple actions, consisting of movement, changing configuration, and adjusting survey sensors. The threads are synchronized through a monitor. The monitor ensures that the randomness of process scheduling does not cause one blob to get ahead of another Blob in steps.

3.1.1 Sensor Model

Mobile robots building a representation of their environment with sensors is a complicated, but well studied task. A common method is to measure return time on laser reflections, which gives a distance to the object it reflected off of. With enough measurements, you can form a range image. This is known as LIDAR (**Li**ght **D**etection **a**nd **R**anging). Using multiple range images at different angles and positions, a composite map can be created that estimates the heights and slopes of the terrain seen [11]. LIDAR sensors have been successfully used in many real world applications to map unknown terrain, and then use that data to navigate while avoiding obstacles [14].

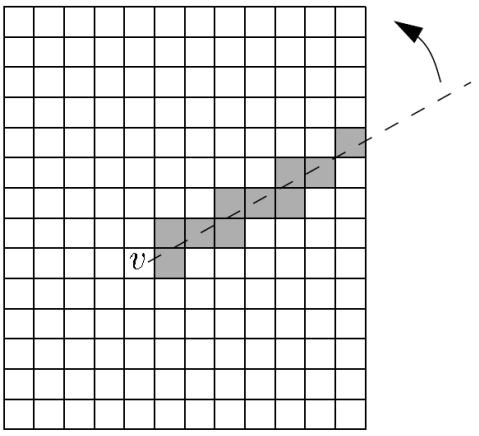


Figure 3.1: Rotating a line over a grid. Gray pixels are in Active structure [9].

To have an accurate simulation, a blob should have a realistic yet practical simulation sensor model. The simulated sensor used can properly detect the height and slope of cells within a limited line of sight view shed from the blobs position. This is a justifiable assumption, as current robotic systems use systems that give similar end results [6]. It is also practical, because there are relatively efficient ways to calculate line of sight on a height map. In particular, van Krevald's line sweeping algorithm has a complexity of $O(n^2 logn)$ for an $n \times n$ grid [7, 9]. In this algorithm, a line segment with length equal to the max distance being considered is rotated around the viewpoint by the Z-axis. (See Figure 3.1) A structure known as the active list stores the cells in the height map that intersect that line, which are ordered by distance to viewpoint. The active cells are processed in order, and a cell is said to be visible if the gradient from the viewpoint to the cell is greater than the gradients seen earlier in the active cell list.

There are just two parameters in a simple line of sight sensor. The maximum visible range, and viewpoint height above the terrain. In both

cases, they are directly related to the modules contained in the Blob, both normal and survey modules. They are modeled off the expected shape of the blobs. Assumptions are made, and these values are calculated being based on if the blob was in configurations like simple geometric shapes. The view height comes from equations related to a volume of a pyramid, and the view distance from volume of a sphere.

The number of generic modules will be represented as n_{mod}. The number of survey modules will be represented as n_{surv}. A survey module is considered to be the size of 5 generic modules. The maximum view distance for a blob is set as $v = 5 * \sqrt[3]{\frac{n_{mod}}{\pi}}$. While the constant 5 is arbitrary, the cubic root ratio is chosen because that is how the radius of a sphere increases as its volume is increased. Only the generic modules along the edges of a blob can use their sensors. The survey sensors being carried are assumed to not be a type of sensor useful for determining terrain information. The formula used to determine the viewpoint height is $h = \sqrt[3]{3 * (5 * n_{surv} + n_{mod})}$. This is the formula for finding the height of a square pyramid based on its volume, where its base lengths are the same as its height. The survey modules are included as they would increase the size of the overall blob, and therefore its viewing height An example of this line of sight view is in Figure 3.2. In each of the images, the areas not blue tinted are visible to the blob at that time. The brighter the areas, the higher up they are.

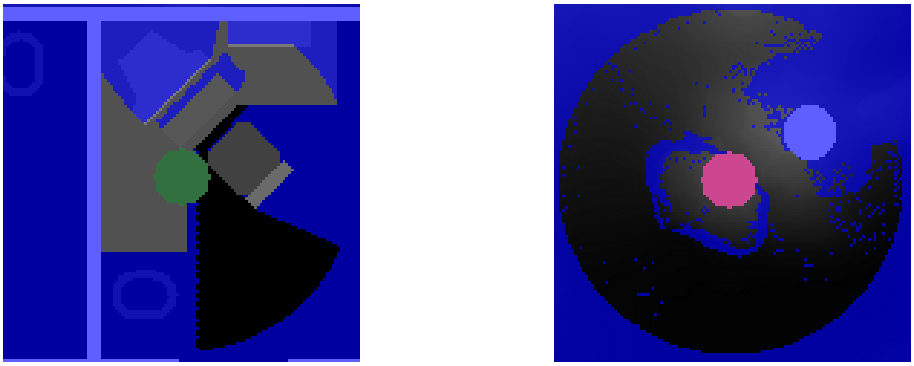
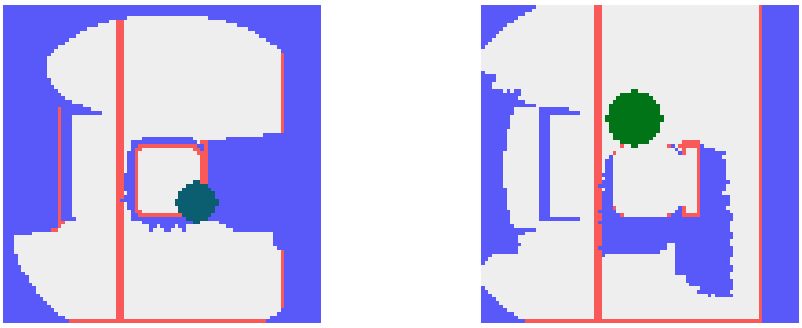


Figure 3.2: Line of sight view sheds overlaid on grayscale height maps (Blue Tinted = not visible)

3.1.2 Movement Model

A blob's movement model determines which cells are traversable or not, based on both the cell's properties and what the blob contains. It also determines the cost for traversing that particular cell. A relatively simple movement model is used for this simulation. From the height map, a slope map is constructed using LandSerf, a free Geographical Information System [19]. If a slope of a cell is below a certain threshold, a blob can traverse it. This threshold is related directly to the number of modules in the blob. The threshold used in this work is $\sqrt{2 * (n_{mod} - 5 * n_{surv})}$. The survey modules have no capability to move on their own, so it requires extra normal modules to move them around. It is true that in this movement model, more modules is always better. This might not be true for all SRR systems though. In some cases, larger blobs might take longer to move around despite being able to accomplish more. Smaller blobs might also be more agile and able to reach more cramped areas. For the purpose of this simulation, the movement model is kept relatively simple. In the example in Figure 3.3, the smaller blob is stuck on top of a steep platform, yet the larger blob has enough modules to climb up and down that same platform on its own.



(a) Size = 400 (b) Size = 1000

Figure 3.3: The ability to traverse the same terrain differs in different sized blobs. (Red = impassable, Blue = Unexplored)

3.1.3 Communication Model

Without some sort of wireless communication, there can be no cooperation between blobs of modules once they are separate from each other. To simplify the simulation, an assumption is made that communication is faultless and immediate. In a real scenario, communication is not guaranteed, but that aspect of mobile robotics is not the focus of this study. Additionally, the distances simulated amongst the robots are not thought large enough to cause significant signal disruption if it was taken into account. A blob can both broadcast a message to all blobs or direct it at a single blob. Messages contain the sending Blob's ID, the type of message, and the payload related to that. Messages are received instantly by the other Blobs, and multiple messages can be sent back and forth during a single simulation step. There are a few message types that are sent between blobs, and they are listed with brief descriptions in Table 3.2. More detail on how each are used is in section 3.2.1.

3.2 Control System Design

The blobs are controlled by a system based off the ALLIANCE architecture. "ALLIANCE is a software architecture that facilitates the fault tolerant cooperative control of teams of heterogeneous mobile robots performing missions composed of loosely coupled subtasks that may have ordering dependencies" [12]. It is a distributed architecture, designed to respond well to failure of team members completing their tasks, along with noisy sensors and communication. It can be applied to Self Reconfiguring Robots, but there are areas it does not take into account. In ALLIANCE, the addition and removal of robots within the system is beyond its control. Robots are only added by humans manually, and are removed either manually or by failure of a system. In SRRs the number of unique agents at any time is not static. Instead, the agents modify their own numbers through splitting apart or merging together.

Within a ALLIANCE control system, there are layers of behaviors. The lowest level behaviors are simple like movement and obstacle avoidance. Higher level behaviors are more complex, like controlled exploration or manipulating other objects. Higher level behaviors can suppress or use lower level behaviors. Of the highest level behaviors, only one of them can be active within a robot.

Type	**Payload**	**Description**
CHANGE POS	Point	Contains the current location of the blob.
CHANGE SIZE	int, List	Contains new size of Blob. Both number of generic modules and list of specials.
REMOVED	None	Notifies others that this blob is destroyed.
CREATED	Point	Notifies others that a new blob is created, contains its position.
EXPLORE TARGET	Point	Notifies other blobs it is exploring a location.
MAP DATA	Map	Shares its map data with other Blobs.
ASK HELP	Point	This blob asks for help, sharing its current position. Broadcasted to all blobs.
CONFIRM HELP	Point	Blob confirms it can help another, sharing its position. Sent to the particular blob that asked for help.
MODULE PLACED	Module	Sent when a Survey Module is placed, with its location and sensor attributes.
PLACING SENSOR	Point, float	Tells other blobs it plans to place a sensor at a location, and how far away it is from that point.

Table 3.2: Communication Messages.

There may be multiple high level tasks a robot is programmed to do, but only one can be active at a time. It determines which task is active by assigning a motivation to each one. The motivations are modified by sensory input, other robot communication, and internal excitatory and inhibitory functions. The two main types of functions are called *robot impatience* and *robot acquiescence*. Acquiescence lowers motivation for a robots current task if it detects it is failing to accomplish that task. This allows the robot to give up on tasks that it is unable to accomplish or having difficulty with. Impatience deals with knowledge of other robot's tasks. If it knows another robot has been trying to accomplish a task, but it is failing to make progress, its motivation for that task will rise. It can then take over that task from the failing robot if motivated enough. One aspect of the ALLIANCE architecture, is that while it is well adapted for unexpected failures, it assumes that none of the agents will deliberately try to fool other agents into believing something false. This assumption does carry over into our implementation.

3.2.1 Implemented Architecture

The implemented system draws much from the ALLIANCE architecture. It has a three layered architecture. The lowest level deals with sensor data and receiving messages, as well as performs the set of basic actions in Table 3.1. The middle layer deals with obstacle avoidance and path planning. The highest layer are complex tasks, which are selected by their motivations, such as exploring the map or deciding when to split. Their motivations are based off data from the lower layers, as well as previous states. The lowest level integrates all sensor data together into an existing map, tracking what areas have been explored and not. This map, known as a vision map, stores the discovered height and slope information, and is available to the higher levels. Each Blob uses two movement models, which is uses to create two obstacle maps, based on what cells are traversable. The first tells which cells are impassable by the current blob, and the second knows which cells are impassable by the entire system. Higher tasks can use this information to avoid difficult areas.

All incoming messages from other blobs are received at the lowest level. Some messages are processed, but all are forwarded up to the high level tasks

unchanged. The CREATED, REMOVED, MAP DATA, CHANGE POS, and CHANGE SIZE messages are all looked at to keep track of what Blobs exist, and their current status. When sensor data from another Blob is received it is integrated together into a global map. Integrating two vision maps in different areas of the same terrain is an easy problem if the relative position of the two blobs is known [1]. The lowest level does not act on its own, it can not move, split, merge, split, deploy or pick up a sensor without orders from a higher level.

The middle level acts on commands from the higher level tasks. A task can tell it to move to a specific location on the map. Using the developed obstacle maps, it uses an A-star path finding algorithm to find the fastest waypoints to that location [4]. A-star is a best-first graph search strategy. At the starting point, metrics are calculated for each adjacent node, which are the distance traveled plus an estimate of how much more to go. Nodes adjacent to the lowest cost node are than tried until the destination is reached. The Euclidian distance metric is used in this case. Unexplored cells are treated as passable so it will attempt to explore the shortest path first. If an path is found to the destination, it will move along the calculated waypoints one step at a time until it runs into an obstacle or reaches its destination. Because it often is exploring new terrain while moving, recently discovered obstacles can make its path invalid. It is also possible that the A-star algorithm can find no valid path to the destination. In all three cases, the blob will notify the higher level of what occured, stop moving, and wait for the high level tasks to re-designate a destination.

3.2.2 High Level Tasks

The most complex algorithms are executed at the highest level through a set of unique tasks, of which only one is active at a time. Each task has a motivation to be executed, and internal state variables. A motivation simplifies down to a float value, limited within a minimum and maximum, which is all in Table 3.4. Since no specific motivation values were recommended in the ALLIANCE papers, a range of 0 to 1 was chosen for simplicity. Neutral and activation were also chosen for simplicity and for approximately splitting the range. A task can only be executed if it is above the activation threshold.

Idle Task	The Blob Idles, and performs no action.
Broadcast Status	Periodically sends status data, including position and map data to the other blobs.
Explore Task	Explore unknown areas of the map. Decides when and where to explore, while cooperating with other blobs.
Split Task	To accomplish multiple goals at once, the blob can decide to split in two. It only splits if it is not having too much difficulty doing its current task.
Help Task	The help task determines if the blob asks another blob for help, or decides to help another blob.
Merge Task	The merge task kicks in once it is decided that two blobs are helping each other. The blobs move towards each other and merge when within range.
Place Sensor Task	When the time is right, the blob will decide a good place to put a survey sensor. Once decided, a blob will move there and place it.

Table 3.3: Description of different tasks

Tasks can also be flagged as complete, and in that case will not be executed no matter the motivation. Otherwise, at each step the task with the highest motivation is executed. There are seven different tasks in this system. They are Idle, Broadcast Status, Explore, Split, Help, Merge, and Deploy Sensor. A brief overview of them is in Table 3.3. A detailed description of what they do is in sections 3.2.2.2 through 3.2.2.8. Each task performs its own unique actions when executed. Each task implements a set of functions which based such as sensor and feedback inputs, affect its internal state. There are seven of these functions which are seen in Table 3.5. These functions are often responses to specific actions in a lower level, like a blob reaching a destination, splitting or merging. They also include external inputs like receiving a message from another blob, or switching to a different task. Only the currently active task can command the lower level tasks to perform any physical actions.

3.2.2.1 Motivations

Status Update	The simulation updates the internal motivations based on task state and blob sensor inputs each time step.
Execute	Execute the task for a step. This is the only function that can command lower levels to perform actions or seek destinations. Only one task can be executed per step.
Task Chosen	When a new task is chosen, all tasks are notified of it.
Comm Received	A message was received from another Blob. Can occur many times within a single step.
Dest Result	Result from lower layer on a set destination command. The destination could be have been reached, completely unreachable, or an obstacle can be detected along the way.
Split	When a blob splits, each task's motivations and state must be cloned and adjusted for the new blob.
Merge	When two blobs merge, both tasks within must merge their motivation values and state.

Table 3.5: List of Task Functions

Minimum	0.0
Neutral	0.3
Activation	0.7
Maximum	1.0

Table 3.4: Motivation Thresholds

While the motivation for a task can be simplified to a single floating point digit, there are many ways it is modified. Each relevant event does so in a different way. Motivations are updated at the start of each time step, and as a result of outside actions. Sometimes the motivation will be set to a constant amount. Often in the case when a blob has completed a task, the motivation for that task is set back to neutral. Another well used method is to just adjust the current value by addition or multiplication. For example, when a sensor detects an obstacle, the Split task's motivation is adjusted to 60% of its current value. A blob does not want to split when it is having difficultly traversing the map. Excitatory and Inhibitory functions can also be set up. These change the motivation linearly or exponentially each step automatically. Thresholds can be placed on these functions, so the function only runs when it is above or below the threshold. As a example, a inhibitory function is placed on the motivation of the Help task with a threshold at neutral motivation. That way, any time its above neutral, the motivation will decrease back down. Singular events raise the

Help motivation in large amounts, but over time the inhibitor brings it back down. Optionally, a flag can be set keeping track of when the motivation was last modified. It will not adjust motivations unless it has not been modified for a certain number of steps. This allows for the effect of actions to persist for temporary amount of time before reset. Adding to the previous example, that inhibitory function could be made so it only starts its effect when it is above the neutral value and has not been modified for 30 steps. Various events increase the motivation to help other blobs. But if enough time passes without any more events occurring, it grows impatient, and the motivation for helping lowers back down towards neutral.

A matrix of motivations is used by some tasks, which can be used to derive motivations for the overall task. It is useful when a task is very dependent on the location in which the task will be performed, such as the Explore task. It has to decide not just when it should start exploring, but what locations to explore as well. Each cell in the motivation matrix is directly related to a square area in the environment. Each cell in the map can be modified independently, in the same ways as a normal motivation. Inhibitor and exciter functions can be set up to affect all cells in the map, but each cell has its own unique last modified flag. This allows the map to have memory of events at different locations. One special function allows adjusting not just one cell, but all cells around it. It uses a Gaussian function of a specified radius centered at a specified location, and changes the motivation of both that cell and the cells around it. Further away cells are changed less then closer cells. In the Exploration task, when another blob says it is exploring an area, the motivation for exploring that cell and all cells around it is decreased using this Gaussian function. This can be seen in Figure 3.6. Because the last modified flag is set in that area, the exciter function on that map does not kick in immediately, giving the map memory of what areas other blobs are exploring. In another function of the map, the motivation map can take a matrix of bits the same size as the map, and modify just the motivation of cells that contain bits set one way. The Explore task uses this to set the motivation for all explored areas to zero, by using a bit matrix of already explored areas. This insures that no area is purposely explored more than once, since a motivation of zero signifies that area is completely explored. The exploration task's motivation map is covered in more detail in Section 3.2.2.4.

3.2.2.2 Idle Task

The Idle task is the simplest of all tasks. When executed, the blob does nothing. It has no internal state variables. Its motivation is constant, and equal to the activation motivation. If no other tasks have above activation motivation, it is selected. It is the default task. None of the task functions modify the motivation or internal state at all, as it has no state to keep

3.2.2.3 Broadcast Status Task

The Broadcast status task is also quite simple. Its purpose is to periodically broadcast information to the rest of the Blobs. This information includes the current position of the blob, and this blob's current knowledge of the environment. Sharing this data allows proper cooperation between the blobs.

Parameters:

Frequency: How often, in simulation steps, to broadcast. In the simulation, this number was set to every 15 steps.

Status Update:

The status update function updates the excite function, which activates after 15 steps have passed since the function last executed. Motivation rises rapidly from zero to the maximum quickly. It rises linearly by 0.2 per step, so after a maximum of 5 steps it will be at maximum motivation. This guarantees the blob will enter the Status Update task every 18-20 steps.

Execute:

When the broadcast task is executed, the first thing it does is merge together all vision maps it has received from the other blobs. Integrating maps from two or more robots together is a easy problem as long as they know their positions relative to one another. [1] The consolidated map is used to update both the obstacle map local to this blob, and the one global to the whole system. Two messages are then broadcasted to all other blobs. First a CHANGE POS message which includes the blobs

current position. The second is a MAP DATA message, with its most current information, so the other Blobs can take advantage of it. The motivation is then set back to zero and the timer reset.

Split:

When a blob splits, a new broadcast task is cloned from this one, keeping all the same parameters as this one to be given to the new blob.

3.2.2.4 Explore Task

Exploration of the environment is the first step to solve the overall problem. Without knowledge of the environment, the SRR system cannot reliably deploy surveillance sensors in anything close to a good location. A particular blob has to figure out two things: When should it start exploring, and what locations should be explored, all while cooperating with the other blobs doing the same thing. To do this, the explore task has a motivation map, which represents the motivation to explore specific locations. Each cell of the motivation map represents the motivation for a 4x4 group of cells in the environment. The overall motivation for the task is based off these cells, and how far the cell is from the blob. Ranking by distance allows closer cells to be explored first which is usually best for efficiency. To insure different robots will explore different areas, a cooperative version of frontier-based exploration is used [1, 20]. A frontier cell is an unexplored cell which is an immediate neighbor of an explored cell. A frontier map is created from the blobs current vision map, as seen in Figure 3.5. Potential exploration targets are selected from these frontier areas. Generally, motivation for exploration in a simulation will start high, as there is a lot to explore as seen in Figure 3.6. It will become more erratic as more is explored, and then trend downwards and as the last of the map is explored, eventually falling below the activation threshold.

Parameters:

Dimension: The width and height of the environment it should explore is all it needs, so it can create a motivation map of that size.

State Variables:

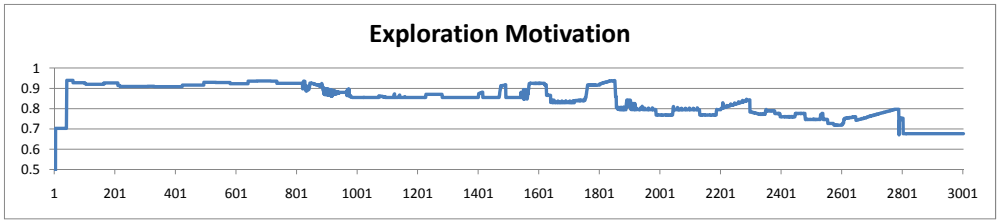
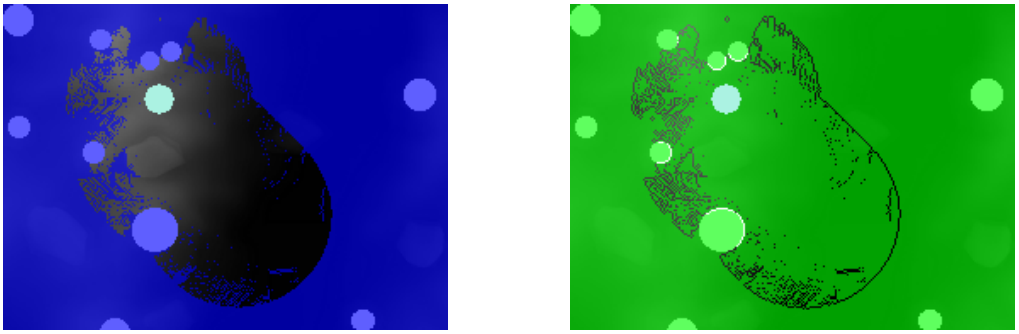


Figure 3.4: Motivation for Exploration trends downwards over the simulation, until it ends up below activation.



(a) Vision Map (b) Frontier Map

Figure 3.5: A Frontier Map created from Vision map. Green tinted cells are are not Frontier cells.

Motivation Map: Holds motivation for exploring any location in the environment.

Best Location: The point which has the highest motivation.

Current Destination: The position the blob is currently moving towards to explore.

Wait Time: When other blobs are exploring an area, that area's motivation will be lowered and unable to recover for a limited amount of time. This allows it to hopefully finish exploring before it tries to take over.

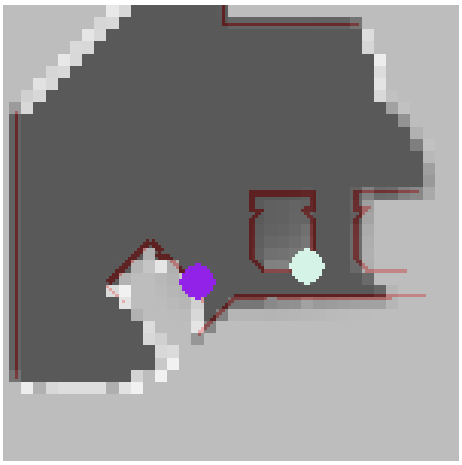
Status Update:

Every step of the simulation, the status update function keeps the moti-

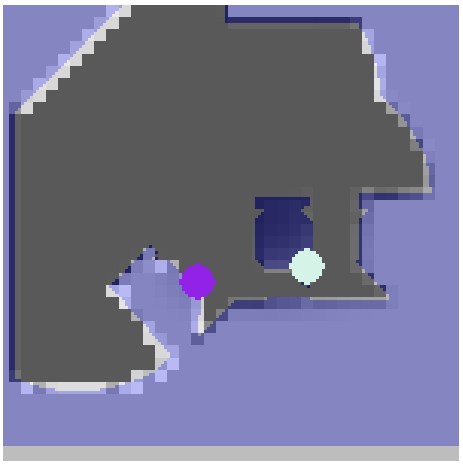
vation map within the explore task up to date. There is both an inhibitor and exciter function on every cell in this motivation map. The exciter function kicks in on any cell below neutral motivation that has not been modified in 50 steps. The motivation will raise linearly by 0.008 each step for older cells. The inhibitor function prevents the motivations from rising above the maximum at 1.0. This allows areas which fall below neutral to recover after some time not being modified. There are four additional steps which modify the motivation map. The final result of this can be seen in Figure 3.6.

1. Using a bit-matrix of which cells are unexplored or not, all cells in the motivation map which have explored cells in the vision map are set to zero. There is no reason to to explore areas twice.
2. Using a frontier map created from the vision map like in Figure 3.5, adjust upward the motivation of cells that contain frontier cells, and have not been modified for 50 steps. Since the frontier cells are the destinations, they are the ones with highest motivation.
3. Using the obstacle map created from the whole blob system's movement model, adjust downward the cell's motivation with obstacles in them. This movement model knows how the blobs would move if all were combined, so these obstacles should be considered completely unreachable. Motivation is kept low to prevent attempting the impossible. It will not affect cells that have been modified within 50 steps.
4. Using the current blob's obstacle map, slightly lower the motivation the cells with obstacles in them. This encourages going for easier targets first, but not enough to prevent the blob from exploring those locations eventually. This also follows the 50 steps rule.

Once the motivation map is updated, the next step is to choose the best location to explore. It only does so if it is the active task and not already pursuing a destination. To find the cell which has the highest relative motivation, it takes all cells with motivation over activation, then scales them based on how far away they are from the blob. A cell within 10



(a) Explore motivation over Obstacle Map (Red = obstacle)



(b) Explore motivation over Vision Map (Blue = unexplored)

Figure 3.6: Explore motivation map overlaid both vision and obstacle maps. Brighter areas are higher motivation. Notice that the frontier areas with obstacles have lower motivation that frontier areas without them.

distance of the blob is not scaled at all. A cell of distance 200 or greater is scaled by 0.9. Cells falling between 10 and 200 are scaled linearly between 1.0 and 0.9. The cell that has the maximum scaled motivation becomes the best location. A random location within the cell is chosen as the exact destination to explore. The motivation for the overall task becomes the scaled motivation of that cell. The small amount of scaling causes it to explore the closest good areas first, but does not discourage exploring farther away areas too much.

Execute:

The execute function only does something when the current destination needs changed. Otherwise, the lower layers take care of movement towards the destination. When a destination was changed, it notifies the middle layer to of the new destination, so it can find a path. Once it does this, a EXPLORE TARGET message is broadcasted to all other blobs containing the position it is exploring. This way other blobs can avoid exploring the same location.

Task Chosen:

If the task chosen is not Explore or Broadcast Status, the Explore task will reset its internal destination. It does this so it will normally update its motivations again. It does not update its motivations when it is active and exploring, so a reset is needed. If the active task changes to a task other than Broadcast Status, that task might set its own destination. This way, if the Explore task becomes active again, it can choose a new destination.

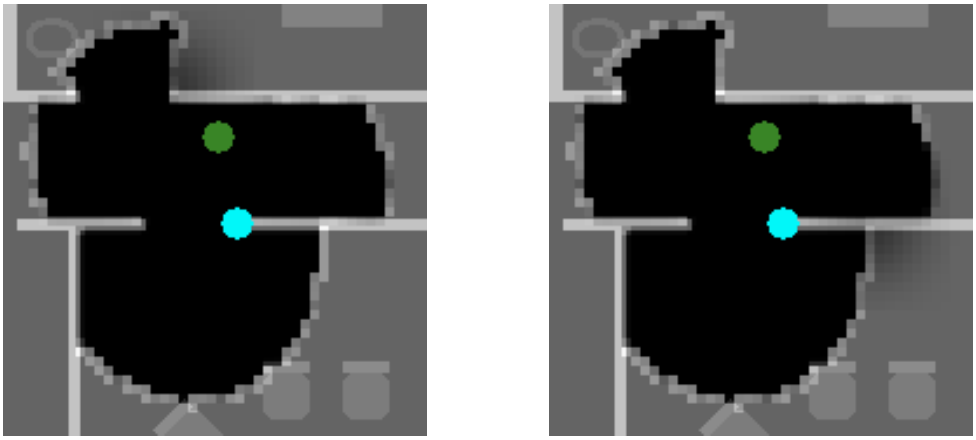
Comm Received

The only message the Explore task pays attention to is to EXPLORE TARGET messages from fellow Explore tasks in other Blobs. Knowing where other blobs are exploring prevents multiple blobs from exploring the same area. When it hears another blob is already exploring an area, its own motivation in that area is decreased. A Gaussian function cuts the motivation in half at that location, with decreasing changes out to a 35 cell radius. The motivation's of the affected cells stay low for at least 50 steps. This gives that blob some time to finish exploring its planned area. If it fails to explore it within reasonable time, another blob might grow impatient, its motivation for that area will rise, and it can take over the job. In Figure 3.7, two blobs have decided to explore two different areas, so the area the other is exploring has been darkened in each of their motivation maps.

Dest Result

The explore task only pays attention to destination results from the middle layer that originated from itself. There are three different results that can be returned.

Obstacle Detected: When a obstacle is detected, it updates the obstacle maps, to ensure they are up to date with new information. It then resets the destination to the same value, so next step the execute function will run again to find a new path to the destination, hopefully bypassing the obstacle.



(a) Lower Blob's Mot Map (b) Upper Blob's Mot Map

Figure 3.7: Cooperation in Exploration. Brighter areas have higher motivation.

Destination Reached: When the destination is reached, it will reset its internal destination to null, allowing the task to update its motivation normally again, and choose a new destination to explore.

Destination Unreachable: When a destination is found to be unreachable, it also resets the internal destination to null, but it also temporarily adjusts the motivation for that surrounding area downwards significantly. This is so when it tries to find a new place to explore, it doesn't immediately pick the same one, and tries something new. This effect on motivation is much more temporary then when a different blob tells us its exploring an area. In this case, the motivation will start to recover immediately due to the excite functions.

Split

When another blob is split from this one, it must create a new Explore task for it. In this case, it receives almost an exact clone of the task, which includes the motivation map. To ensure cooperation, both explore destinations are reset to null. This way they will recalculate their individual motivations and go separate ways.

Merge

When two blobs are merged, they must consolidate their motivations and views of the environment. In the case of the two explore tasks, the motivation maps are merged together cell by cell, setting the merged value to the maximum motivation value between the two. That way, if one blob had not finished an exploration goal before merging, the new blob should see that area as clear for exploration again.

3.2.2.5 Split Task

The split task's purpose is to decide when this blob should split into multiple blobs to be able to better distribute its work. Ideally, a blob won't split if that causes the child blobs to be unable to traverse the terrain efficiently. Since this is dependent on the environment, and that is unknown at first, this has to be estimated as it is exploring. It will not split when it is under a minimum size, which is adjusted based on the difficultly it is having when traversing the map. When it does decide to split, it does so into two equally sized blobs.

State Variables:

Min Split Size: The minimum size a blob must be to consider splitting. Initially is 300, but is lowered if the terrain seems easy, and increased if the terrain seems difficult.

Obstacles Detected: The number of times it has detected an obstacle since successfully reaching a destination.

Destinations Reached: The number of times a blob has reached a destination successfully without running into a obstacle.

Wait Time: Time to wait before increasing motivation to split after an event. Initial is 40 steps.

Status Update:

There is both an excitatory and inhibitory function active in this task. When the blob's size is greater than or equal to the minimum split size, the inhibitor function is disabled. Instead, a linear exciter function is

enabled, which will increase the motivation by 0.01 each step if there has not been an obstacle event in the last 40 steps. It will raise the motivation until the maximum motivation value. If the blob size is below the minimum split size, the exciter function is disabled. Instead, an inhibitor function in enabled which caps it to neutral motivation. The inhibitor function can be seen rising the motivation frequently in Figure 3.8. A blob will only try to split if it's above the minimum split size.

Execute:

When the split task is executed, a new child blob is created using some of its modules. It splits both normal modules and special survey modules evenly if possible. For odd number of modules, the parent blob receives the extra one. After the split, it adjusts the split motivation to halfway between neutral motivation and minimum motivation, and sets the modified flag. This prevents the exciter functions from raising motivation until the wait time has passed. A CHANGE SIZE message is then broadcasted to all the blobs, consisting of the new size of the parent Blob. The new blob on its creation notifies all other blobs of what occurred using a CREATED message. In Figure 3.8, splits have occurred at steps 102, 220, 984, 1718, and 2685. In that example, the blob is navigating a difficult map. Because of this, blob always merges soon after it splits.

Task Chosen:

If the task chosen is the Split task, the complete flag is reset to false. This allows the task to be executed again. If the task chosen is Merge, the motivation is set to neutral, and the modified flag is set. This is to prevent the Blob from splitting soon after it merges, or even while its attempting to merge. If the map was found difficult enough to require a merge, it should not be splitting apart even more.

Comm Received

When a CONFIRM HELP message is received, it will reset the split task motivation back to neutral. These messages are sent by the merge task occasionally, to keep merging blobs updated on each others positions. Resetting the motivation keeps a blob from splitting up as its on route

to merge with another blob. In Figure 3.8, there is a rather long merge around steps 650 - 900. Throughout this merge, the split motivation is reset periodically to prevent the blobs from splitting during a merge.

Destination Result

The Split task uses the feedback on how well other tasks are doing to determine its motivations. Generally, if other tasks repeatedly run into obstacles when trying to execute, its willingness to split will go down. Also, if other tasks successfully get to their destinations repeatedly without difficultly, its willingness will be increased. Two variables referred to earlier are kept track of. The number of Obstacles detected since the last time it successfully reached a destination, and the number of destinations is has reached, without running into an obstacle.

Destination Unreachable: If any task's destination is found to be unreachable, the Split task's motivation is lowered by half, and the modified flag is set. Once a potential goal is found unreachable, it will not consider splitting again for some time.

Destination Reached: When a destination is reached, the motivation for the Split task does not change. Instead, if the number of destinations reached without detecting any obstacles reaches six, the minimum split size is reduced to 80% of its former value and the counter is reset. This allows the task to adapt to easier environments, by splitting into even smaller and more numerous blobs.

Obstacle Detected: If an obstacle is detected, the motivation for the split task is adjusted downward. Depending on how many obstacles were detected since the last reached destination,O_{dec}, the motivation is adjusted downward by $1 - (O_{dec} * 0.1)$. O_{dec} can be a max of 5. For obstacles, the adjustment to the motivation does not set the modified flag, so it's only a temporary setback for the task. In Figure 3.8, between steps 280 and 330, multiple obstacles are detected, stopping the rising motivation and bringing it way down. At step 330, the destination is found to be unreachable, so the motivation is lowered by half, and stays constant for a short while.

Split

The execute part of this task actually splits the blob. When this task is notified of the split, it needs to clone itself, copying its current minimum split size and wait time to the new Split task of the new blob.

Merge

When two blobs merge, their split tasks must also merge. Of the two minimum split sizes, the maximum of the two is kept. If after they merge, the new size of the blob is bigger than the minimum split size, the minimum split size is increased by a factor of 1.25. It does this because since the blob found the map difficult enough to need to merge, the minimum split size is probably too small for it to effectively complete its task. In very difficult maps, the system may go through several split/merge cycles until it finds a reasonable size which balances traverse ability with parallelism. In Figure 3.8, it can be seen that the minimum split size is increased each time the merge task ends, meaning a successful merge. This was a very difficult map, so the minimum split size continues to rise each time the blob merges, until its eventually greater than the total number of modules in the simulation. At that point, the SRR is effectively a single traditional robot.

3.2.2.6 Help Task

The Help Task determines when a blob needs assistance with its goals, or if it is willing to help another blob out. In general, the motivation to help will increase as a blob has difficultly achieving its goals, and decrease if it is not having difficulty. The motivation is also increased when other blobs ask for help. Blobs help each other by merging together. Blobs must ask each other for help and confirm that help before they can continue to the merge task.

Parameters:

Help Weight: A variable which changes how willing a blob is to help other blobs, or ask for help itself. This is one of the parameters

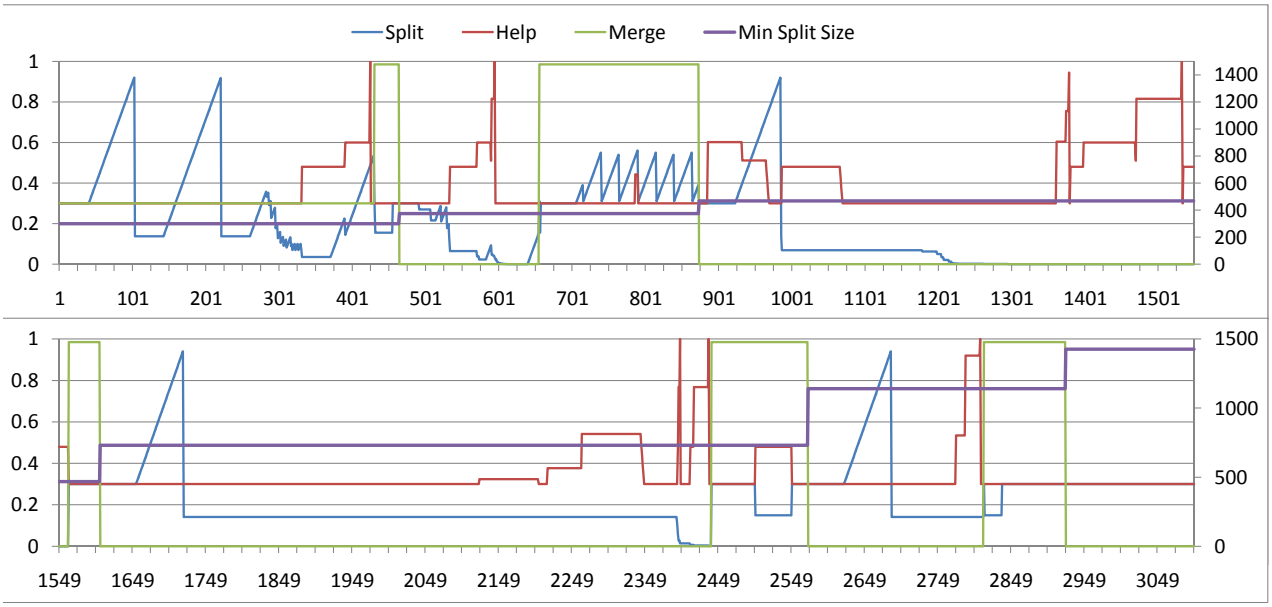


Figure 3.8: Blob 0, Office map. Graph of Split, Help, and Merge motivations, as well as minimum split size.

which varied in each experiment. It is tested at values of 0.5, 1.0, and 2.0.

Help Radius: The maximum radius from which this blob will consider helping another blob. This is kept constant throughout the simulations at 500.

State Variables:

To Help: The Blob it is currently thinking of helping.

Status Update:

First, the motivation for the Help Task is capped at the maximum. Also, an inhibitor function is in place, so that if the motivation is above neutral, and is unmodified for 80 turns, it will decrease linearly back to neutral. This allows the blob to become disinterested in helping or getting help if nothing happens for a while. If the inhibitor function causes the motivation to drop below neutral, it will reset the To Help blob,

which was the blob it was thinking of helping. That way, after a significant time passes without further incident, it can forget everything that has happened. In Figure 3.8, around steps 1060 and 2370, there are examples of help motivation above neutral dropping down after a period of no change.

Execute:

There are two cases that can occur when this task is executed. It either has been asked by another blob for help, or it has decided it needed help on its own, and will be asking for help from others. If the To Help variable is not set, it broadcasts an ASK HELP message to all blobs. The message contains the blobs current position. If the To Help variable has been set to some blob, it sends a CONFIRM HELP message just to the particular blob that had asked for help. That Motivation for this task is then reset to neutral afterwards. In the example, at step 594, 1533, 2397, and 2436, the blob is asking other blobs for help. At steps 424, 1378, and 2807, the blob is confirming it is willing to help another blob.

Task Chosen:

The help task does change anything with its internal state when other tasks are chosen.

Comm Received

The two messages the Help task pays attention to are ASK HELP and CONFIRM HELP. When it receives a CONFIRM HELP message, this means another blob has answered our request for help, so the motivation for help is lowered back to neutral. Multiple CONFIRM HELP messages could be received at once after it asks for help, but nothing else happens except the lowering of motivation. The Merge task deals with merging the blobs once they state they are willing to help each other.

If it receives an ASK HELP message, it must evaluate how to respond to this broadcast from another blob. It first sets the To Help variable to the blob that sent the message, so it knows who it might be helping. The message will increase its motivation, but how much depends on the distance to the other blob, the help weightH_{weight}, and the current task.

If the distance to the blob is above the maximum help distance, there is no change, but otherwise, the formula below gives exactly how much the motivation is adjusted by.

$1 + (T_{deploy} * H_{weight} * (1 - \frac{dist}{H_{rad}}) * H_{max} - 1)$

H_{weight} is one of the independent variables tested, which varies from 0.5, 1.0, or 2.0. The test varies how willing blobs are to helping each other. H_{rad} is the maximum radius a blob will help at, which is set constant at 500. Practically, there is no maximum, due to the environments which our tests being run being smaller then 500 cells across. T_{deploy} varies between two values. It is 1.6 if the current task is DEPLOY SENSOR, and 1.0 otherwise. This makes it more likely to help out while deploying a sensor, then doing another task like exploring. This is because deploying sensors is considered higher priority. H_{max} is set at 2.2, and is constant. It would be the maximum change to the motivation if the blob that wants help was a distance of 0 from this blob. As it gets farther away, the effect of it decreases linearly, until it is zero at the maximum distance. In Figure 3.8, the spike at step 423 is due to Blob 1 asking for help.

Destination Result

Destination Reached: If the motivation is above neutral when a destination is reached, the motivation is adjusted downward to reflect that it is proceeding well with its task and does not need help. The motivation is adjusted to to 85

Destination Unreachable: When a destination is found to be unreachable, the motivation to help or be helped is increased. Specifically, it is adjusted by $1 + (H_{weight} * T_{deploy} * 0.6)$. The variables are the same as in the previous section.

Obstacle Detected: When an obstacle is detected, it only adjusts the motivation if it was last modified more than 4 steps ago. This lowers the ability for a blob in the process of exploring a new area to rapidly increasing its motivation if it discovers many obstacles in a short period of time. When it does adjust its motivation, it is adjusted upwards by $1 + (H_{weight} * T_{deploy} * 0.25)$. Around steps

2400 in Figure 3.8, a series of repeated obstacles and unreachable destinations creates a sharp spike in the motivation.

Split

When the blob is split, the new Help Task is cloned with the same maximum help radius and Help weight.

Merge

When two blobs are merged, the motivation of the surviving blob is set back to neutral, with the modified flag set.

3.2.2.7 Merge Task

When a blob asks for help, and one or more blobs confirm they are willing to help that blob, the Merge task picks one of the answering blobs, and sends its own CONFIRM HELP message to it. This will initiate a merge. When merging, both blobs will move towards each other, periodically updating each other of their positions. When they reach each other they perform a merge. While it is important to note that it is assumed no blob will intentionally try to fool another, the merge task can timeout waiting for confirm help messages if one blob decides it doesn't want to merge anymore. Motivation is handled differently for the merge task than most other tasks. The task is either disabled, where is stays constant at neutral motivation, or it is enabled and constant at a very high motivation. Specifically, this is $Mot_{act} + 0.95 * Mot_{max} - Mot_{act}$, or 0.985, very close to the maximum of 1. Once a merge is initiated, it is carried out with this high priority. This priority is chosen in relation to average motivations of the Explore task. It will rank merging as higher priority unless there is a very close and easy location to explore.

State Variables:

To Merge: The blob it is currently merging with.

Merge Position: The last known position of this blob.

Step Started: When it received the initial CONFIRM HELP message.

Confirm Frequency: The frequency at which blobs that are merging will send CONFIRM HELP messages to each other. This is set to broadcast every 25 steps.

Status Update:

There is just one inhibitor function updated in the status update function. If the motivation for merging is above neutral, and it has not been modified in 4 times the confirm frequency steps, (100 steps in this case), the motivation will rapidly be inhibited back to neutral. This allows blobs to exit the merge task gracefully if the other blob is has become preoccupied. When the motivation does fall back to neutral, this blob has grown impatient and the To Merge blob and current Merge position are reset to null. The previous blobs it was trying to merge with are forgotten.

Comm Received

The Merge task only is affected by CONFIRM HELP messages. There are three valid ways it could be receiving these messages. This blob could have asked for help, and one or more blobs are confirming they are willing to help. This blob could have sent its own confirm help message out, and is receiving a confirm back. The last valid instance is once two blobs have both confirmed they are helping each other, they will continue to periodically send confirm help messages to update each other of their positions. Depending on the current state, it treats the message three ways.

1. If To Merge is null, meaning this is the first blob to commit to help, it sets To Merge and the merge position to the values in the message.
2. If To Merge is not null, and the blob in To Merge is equal to one just received in the message, this means that the blob is updating this task on its position. The merge position is updated to the new value.
3. If To Merge is not null, and the blob in To Merge is not equal to the message, then multiple blobs have confirmed they are willing to

help. In this case, the distances from this blob to its two potential helpers are compared. Whichever is lower, that one it kept and the other is discarded.

Unless the CONFIRM HELP message was received from a blob farther away then its current one, the motivation for the merge task is set at 0.985, as discussed earlier. This keeps the Merge task with high motivation, allowing it to complete the merge. Whenever the To Merge variable is changed, Step Started is set to the current step. Before it executes, this task will wait a short while after this to allow other blobs to give it CONFIRM HELP messages to consider

Execute:

When a blob asks for help, it does not immediately start heading for the first blob to confirm its willingness to help. Instead, it will have a short timeout after it receives its first response, to give other blobs a chance to respond. Once it is sure that no other willing blobs are closer, it will start the merge process. It first sends a CONFIRM HELP message to the closest blob that sent it a message. Then, it sets the destination of the lower layer to the last known location of this blob to start moving towards it. When new positions come in from the periodic CONFIRM HELP messages from the other blob, it will reset the destination. When the two blobs get below a distance of 2, the blob with the lower ID will initiate a merge onto the other Blob. The blob with lower ID will transfer its modules over to the other and be destroyed, broadcasting out a REMOVED message. The surviving blob sets its merge Task's motivation to neutral, and broadcasts a CHANGE SIZE message to notify the other blobs of its new modules. The Merge task executes and completes successfully four times in the example motivation graph in Figure 3.8.

Task Chosen:

No changes in the internal state occur when it is notified a task is chosen.

Destination Result

When an obstacle is detected, the merge position is reset, so that next time the task is executed, it will recalculate a route to that location again.

When the destination is found unreachable, or it is reached, it will do nothing. This means the task will wait until it gets a new CONFIRM HELP message before it will start moving toward the other blob again. Reaching its destination might also mean the blob will merge next time the task is executed.

Split

When a blob splits, the Merge task clones itself, but does not clone any of the state variables to the new task. Only the original blob keeps its memory of a blob to merge with.

Merge

When the blobs do merge, the merge task sets its motivation to neutral, and sets To Merge and the merge positions to null.

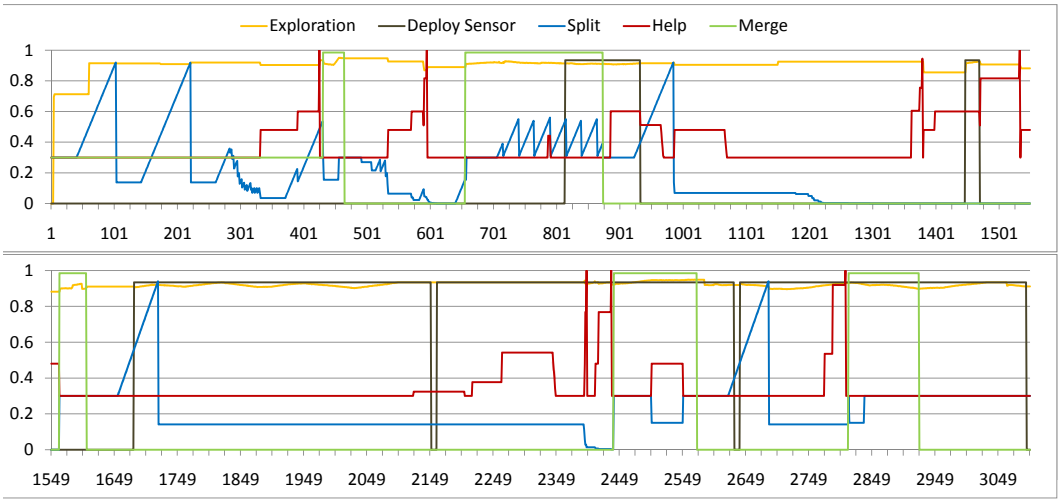


Figure 3.9: Motivations for Blob 0's tasks on a difficult map, missing Broadcast Status and Idle Task motivations.

3.2.2.8 Place Sensor Task

This task's purpose is to decide when and where to place its surveillance sensors. If this blob has no sensors within it, it can't execute this task. Because

deciding a good location for a sensor depends on good knowledge of the terrain, its execution depends directly on what the team of blobs has already explored. The task will periodically compare what has been explored to what it estimates is already surveyed. When it decides enough unsurveyed terrain is explored, it will try to place a sensor. A greedy based approach is used for choosing sensor locations, similar to the max average approach taken in other papers [5]. The big difference in this case, is that the blob does not have complete knowledge of the map. When a chance to choose a location to place a sensor comes up, it chooses the location that will give it the best coverage at that time, given its limited knowledge. When one blob decides a good location for a sensor, other blobs closer to that location can take over the placement, given that they have sensors to place of their own.

Parameters:

Width: The width of the terrain it is exploring.

Height: The height of the terrain it is exploring.

Unreachable Cells: A estimate of the number of completely unreachable cells in the map. It is used for estimating when to place a sensor.

Explore Threshold: A threshold for when this task decides to place a sensor. This is from 0

Status Update Frequency: Kept constant at 80. Only every 80 steps will the Place sensor task recalculate the estimated surveyed map, and consider placing a sensor.

Activation Motivation: Motivation for Place Sensor Task is zero when inactive, and constant at 0.934 when active. This number is also related to a general average of what Explore Task motivation is usually. So when explore Task motivation is high, it will choose to explore over placing a sensor, otherwise it will place the sensor.

State Variables:

Motivation Map: For each location in the map, it determines how good that location is for placing a sensor. Areas that are estimated to have good coverage of the terrain will be ranked higher in motivation then others.

Planned Sensor Locations: Blobs will notify each other when they decide to place a sensor. The task stores these locations and takes them into account when calculating the surveyed estimate.

Surveyed Estimate Map: A vision map built estimating what is surveyed by deployed and planned to be deployed sensors.

Sensor Destination: Current position this task has decided to place a sensor.

Status Update:

The full status update is only run every 80 steps, and the step it executes on is staggered by the blob's ID, so all blobs don't all execute at once. The status update is only run if this blob still has survey modules left within it. If it this is true, the task will potentially go through 4 steps to find where to place a new sensor.

1. It first estimates the areas that are already surveyed. It does this by using a specialized sensor model to estimate what a single survey sensor will see. This estimate sensor model uses the same height and radius parameters as the actual sensors, but has two key differences. First, the sensor model is not given the entire map, but just this blob's current explored terrain. Any cell not explored will be considered unsurveyed by the sensor. Second, while it is a line of sight sensor, it is a much lower resolution version of one. This is to better simulate non-perfect knowledge that a real system may expect and decreases processing time. It uses a resolution of 5, so only one in every 25 cells are checked during the line sweeping algorithm. An example of this estimate compared to the real surveillance can be seen in Figure 3.10. There are some issues with the estimation. In that example, it estimates that the sensor can see through walls northward of its location, and by treating unreachable terrain

as opaque, prevents it from seeing some terrain it has discovered eastward. This is because the wall width is less than the survey resolution, so sometimes it is missed. The limited sensors do give reasonable approximations in a majority of cases though, so it was decided to be acceptable. When it does calculate the estimated view sheds for both the deployed sensors and the currently planned sensors, it meshes them together into the total Surveyed Estimate map.

2. The next step is to find out what is currently explored, but not surveyed. To get the cells explored but not surveyed, the Surveyed estimate map is negated, and logically ANDed with the blob's current vision map. The number of estimated cells explored but not surveyed in that map will be n_{xns}. The number of estimated cells surveyed is n_{surv}. Using the width(w), height(h) and unreachable cells(unr) in the map given in the parameters, the ratio of explored but not surveyed cells is calculated as $\frac{n_{xns}}{w.h-unr-n_surv}$. This ratio, is compared directly to the threshold that is given as a parameter, which is either 25, 50, or 75 in the experiments. The blob must have a certain amount of terrain explored that is not currently being surveyed before it considers placing another sensor. If the ratio is above the threshold, it proceeds to the next step. Otherwise the status update is complete.

3. When enought terrain is explored, it must decide a good location for a new sensor. To do this, it looks at the benefit of placing a new sensor at as many locations as possible, and greedily chooses the best one. It uses the motivation map for this, using a resolution of 10x10 cells. Only one out of every 100 environment cells are checked for the sake of efficiency. It first sets the motivation of all cells in the map that are not visible or completely unreachable to zero as those are obvious bad locations. Next, it goes through each 10x10 cell and picks out the highest elevated cell in the environment that is visible and reachable. It calculates the estimated viewshed at that point with the same estimated sensor model used to create the surveyed estimate map. This viewshed is meshed with surveyed

estimate, to create an estimate of what would be surveyed if a new sensor was placed there. The motivation map value of that cell depends on how much the new view shed and placed sensor cover. The more overall coverage, the higher the motivation for that cell is. After all cells are calculated, the cell with the highest motivation is chosen.

4. The highest point in the chosen cell is selected as the final sensor placement, and the new sensor location is added to the planned sensor location list. The task's motivation is then set to the activation motivation of 0.934. A PLACING MESSAGE is then broadcasted to all other blobs. This message contains the location to place the sensor, and this blobs distance to that location. This blob is not guaranteed to be the one to place the sensor, as other blobs closer to it can take over it. The step this decision was made is stored alongside the location, so it can time out if it can't place the sensor there.

The Place Sensor task does have some acquiescence for its current goal. If it is attempting to place a sensor for a while without succeeding, it can time out. More of the map may be explored by that time, so by doing this the locations are recalculated, it is possible a better location could be discovered. This time out is 6 times the update frequency minus 10, or 470 steps. When it times out, it resets the destination and removes the location from its planned sensor locations. The task grows impatient with other blobs planned sensor locations, at the same rate it grows impatient with itself.

Execute:

When the task is executed, its sole purpose is to set the destination to the location where a sensor is to be placed. This only occurs when a new destination has been chosen, or has been recently reset.

Task Chosen:

When the Place Sensor Task is chosen, it resets the sensor placement location to the same value to insure when it executes, it recalculates the path to the destination again.

Comm Received

The place sensor task deals with two of the messages, the PLACING SENSOR, and the MODULE PLACED message. When a MODULE PLACED message is received, and the module placed was a survey module, it moves that location from its planned sensor locations list to the deployed module list. In the case of the PLACING SENSOR message, it first adds the location to its planned sensor locations. If it is not currently placing its own sensor, it compares its own distance to that location to the distance in the message. If its own distance is lower than the message's distance, and it has survey modules of its own, it will proceed to take over placing the sensor. It does this by broadcasting its own PLACING SENSOR message to other blobs, with the same location, but its better distance.

Otherwise, if its own distance to the location is higher than the one in the message, it will give up trying to place a sensor there. The destination is reset, and the motivation is reset to zero. The lower distance is kept track of, so that requests from other blobs that may be in between its distance and the lowest distance are ignored. For example, Blob 0 decides to place a sensor at a location 300 away, and it broadcasts the message to Blob 1 and 2 which are distance 200 and 100 away from that position. When both Blob 1 and Blob 2 receive the message, they will both attempt to take over the placement, and Blob 0 will give up the location. But once Blob 1 receives Blob 2's message, it will also give up the location.

Destination Result

Destination Reached: When the Place Sensor destination is reached, it then deploys a survey module at that location. After deployment, it broadcasts a MODULE PLACED message to all other blobs so they know of it. It then resets its destination to null and sets the motivation to zero.

Destination Unreachable: When the sensor destination is found to be unreachable, the Place Sensor task takes no direct action. It depends on the help task to get the help needed so that it can

complete that goal. The Help Task is weighted help more during Place Sensor tasks than other tasks.

Obstacle Detected: When an obstacle is detected, the destination is reset so it can plot a new path around it next time this task is executed.

Split

When a blob splits, its Deploy Sensor task is cloned to the new blob. The current motivation map and list of planned sensor locations is copied to the new blob's Place Sensor task. The blob with the lower Blob ID, which is the original blob, keeps its sensor deployment location if it has one, and the child blob's sensor deployment location is not set. This guarantees the blob that continues on with the task has a sensor module, because odd survey modules are always kept with the higher ID blob.

Merge

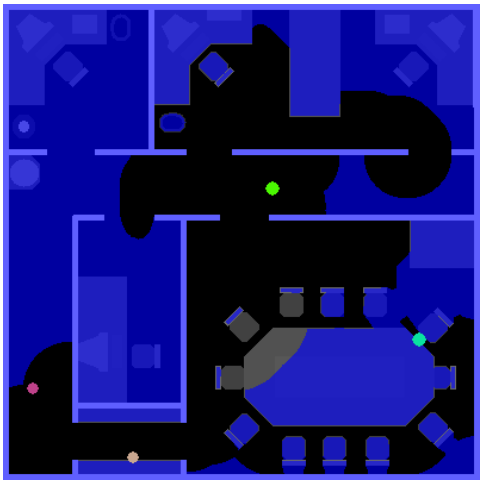
During a merging of two blobs, the blob being merged into takes priority over the one one being merged. If the surviving blob has its own sensor destination, it keeps it. If it does not, it will adopt the destination, motivations, and planned sensor locations of the other blob.

3.2.3 Relations between Tasks

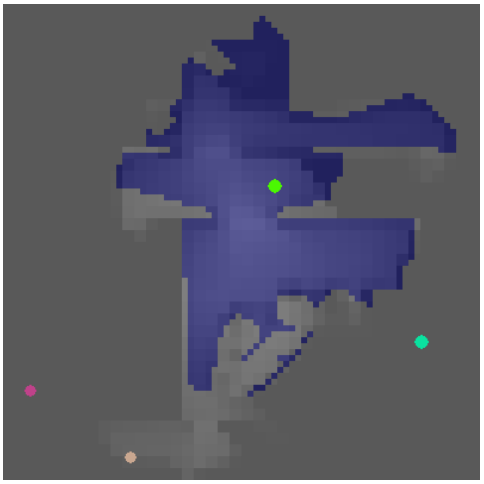
Due to the way the ALLIANCE Architecture works, all tasks within it are closely related. Not only do agents in the system have to prevent individuals from monopolizing a resource or task, but the tasks within each individual must be designed so that no task can monopolize the agents time. This was accomplished by having tasks police themselves, instead of allowing tasks to directly modify each others motivation. In all tasks, there is a clear way to complete the task, which immediately lowers the motivation. In the Broadcast Status, Split, and Help task, it only has one thing to do, so as soon as it is the chosen task, it executes and their motivation immediately is dropped. In other tasks which require movement to a location, (Explore, Merge, Place Sensor), reaching the location causes the motivation to drop. Because reaching a location can fail, all those tasks have the ability to time out or realize the location

is unreachable to prevent the blob from becoming incapacitated. The Merge task requires constant communication the other blob it is merging with. The Explore task will lower the motivation of an area found unreachable, allowing other tasks to take control. The Place Sensor task only has a timeout, as it is designed so that if it cannot reach the area it is trying to reach, it will ask for Help to get there. Only after a significant time has passed will it give up the task.

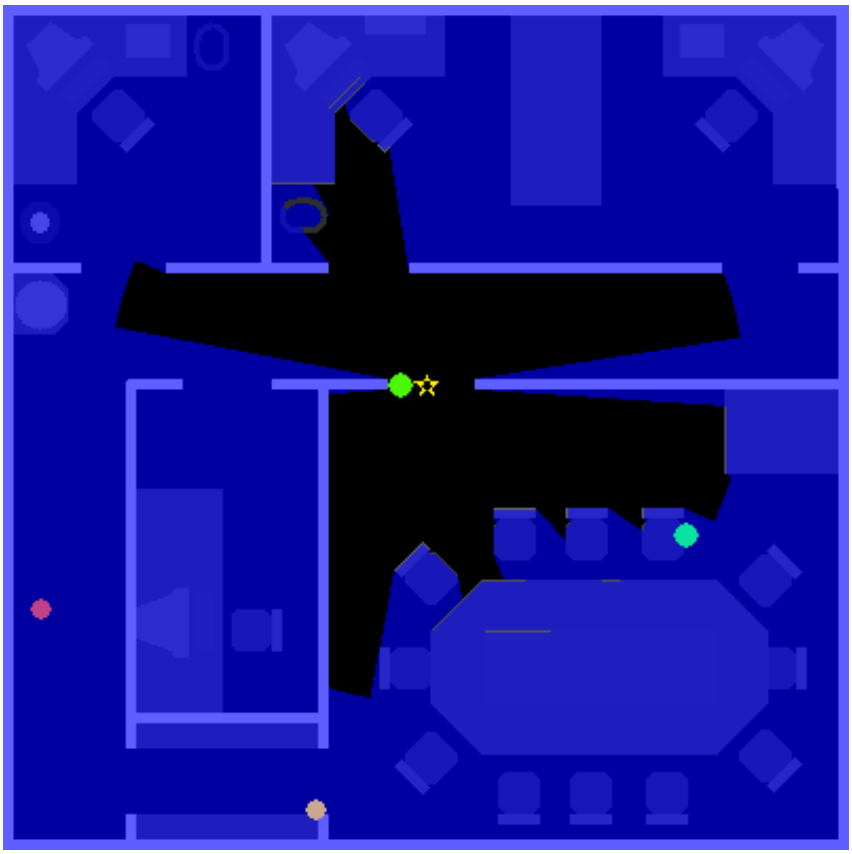
The inter-relation of the tasks goes much further than just preventing deadlock. The 5 main tasks affect and depend on each other in different ways. Only the motivations of Broadcast Status and Idle tasks, aren't based on the Actions of other Tasks. The Help, Merge and Split task are closely related, they work together to delegate out the work. The Help and Merge task can almost be considered one. When the Help task activates, it either starts a merge, or asks other blobs to start one. Both the Split and Help task pay close attention to the progress of the tasks which move the blobs, meaning the Explore, Place Sensor, and Merge tasks. Actions on Help and Split have almost always opposite effects, so when one becomes the active task, the other's motivation drops. Together they help the main tasks of exploring and place sensors overcome difficulties and efficiently parallelize the work. The Explore and Place Sensor task on the other hand, are more of a one way dependency. The Place Sensor task depends on the terrain to be explored before it will try to place a sensor, but the explore task does not care at all where or when the sensors are placed. The motivation of the explore task is based on what itself has currently explored, and what other blobs have sent it from their Broadcast status task.



(a) Explored Terrain when estimate calculated.



(b) Estimated Coverage of sensor before placement.



(c) Actual Coverage of sensor after placement.

Figure 3.10: Comparing estimate of survey coverage to actual coverage. This shows that the sensor estimate does have significant inaccuracies in certain environments.

Chapter 4

Experiment

4.1 Experiment Setup

In order to evaluate the utility of the approach, and test some critical parameters, a series of experiments was carried out using the simulation. An experimental simulation starts with a single blob of modules with a limited number of survey sensors in the middle of the environment. When run, the blob proceeds to move around the map, split, merge and place sensors. The simulation is completed once the last of the survey sensors has been deployed onto the map. Once this occurs, the combined viewsheds for the survey sensors is calculated, metrics are gathered, and all is written to a results file. That particular simulation then ends.

Most variables are kept constant within the simulations. Only 4 different variables are modified. They are the map which the simulation occurs on, the willingness to help other blobs and ask for help itself, and the eagerness for a blob to place a sensor, based on what has been explored. The fourth variable is the number of survey sensors the blob starts with. This variable is tied directly to the map, and is the minimum number of sensors required to survey 90% of that particular map. There are 36 variations of experiments, and each is run at least 20 times to completion. There are 4 different maps, and 3 variations of help weight and sensor placement threshold each. The experimental metrics gathered are averaged together from all 20 runs. By analyzing the results of varying these variables, it is hoped that a better understanding will be gained of how SRR systems function, and whether the approaches used were suitable for the given problem.

4.1.1 Constants

As many things as possible are kept constant during the simulation, a list and description of those follows in Table 4.1. The size of map was chosen as it seemed to give adequate execution time vs. interesting results. The size of the initial blob was chosen at 1400, as that is very close to the number of modules required to traverse to all important areas in all tested maps. The minimum split size is chosen as blobs that size are able to traverse a good portion of all the maps without difficulty.

Number of Blobs	The simulation starts with one blob.
Blob Size	1400 modules.
Blob Position	Start at center of map, at (200, 200).
Minimum Split Size	Initially is 300.
Size of Map	400x400 map.
Timeout	Simulation will timeout after 50000 steps.
Sensor Model	All use the same line of sight sensor model with view height and distance based on size of blob.
Movement Model	All use the same movement model, which determines a threshold slope based on size of blob.

Table 4.1: Simulation Constants

4.1.2 Maps

Four different terrains were used in the experiments. They were developed with an eye on varying difficulty in exploring, traversing, and surveying the map. All maps are the same size 400 x 400, and scaled the same amount. Being grayscale the values in the map vary from 0 to 255. The values are scaled by half in all the maps, so the heights instead vary from 0 to 127.5 modular diameters.

4.1.2.1 Map - Flat

The simplest map is just a flat plain, which is in Figure 4.1. There is no obstacles that should prevent the blob from seeing something or traversing

anything. For a 400x400 sized map, the flat map is expected to require the bare minimum sized blob, number of sensors, and time to complete the goal of surveying the area. Therefore, it is used as a baseline to compare the results on the other maps with.

4.1.2.2 Map - Forest

The forest map in Figure 4.2 is an environment with gentle hills, scattered rocks, and high trees. The trees are completely impassable and block vision, but otherwise it is a relatively easy map to traverse. The highest slope not at the tree's trunk is only 16, so blobs as small as 128 can traverse the entire non-tree map with no issue. The trees and hills make it harder to see longer distances, increasing the number of sensors needed to survey the terrain, and increasing the time it takes to explore.

4.1.2.3 Map - Porch

The Figure 4.3 is a front porch and yard with some trees, with a sidewalk leading to a road. The trees are also completely impassable. Much of the map is relatively easy to traverse, but other parts are quite steep and require most of the blob to be together before it can climb up there. In particular, the railings on the porch require at least a blob of size around 800 to get onto. There are some obstacles preventing seeing larger distances, but less than the forest map.

4.1.2.4 Map - Office

The map in Figure 4.4 models an Office, complete with tables, chairs, computers and a water cooler. It has long flat hallways and open space, but has many steep slopes at the edges of tables. The rooms are cramped, preventing from seeing a large distance, and it is impossible to climb over the walls. To climb onto any of the tables, most of the modules in the simulation have to be together. While they can be split apart while it is exploring the floor, the blob needs to be mostly together to move around the entire environment easily. It is expected that this will be the most difficult map of the four, and that the

modules will have to spend most of its time as a single blob to be able to survey it effectively.

4.1.3 Help and Cooperation

In Fabricant's previous research [2], different behaviors were tested that changed how Blobs help each other. The blobs had a Limited, Generous, Neutral, and Greedy mode. In Limited mode, no splits or merges were allowed. In Generous, it gave priority to helping stuck blobs over pursuing its own goal points. In Greedy, it was the opposite. The Neutral mode gave both equal priority. A similar method is used to test our approach. There are also four different behaviors tested, and one of them is a Limited behavior, with no splits or merges. The Limited mode gives a good baseline for showing that the modular aspects of SRRs are useful. For the other behaviors, because a motivation approach is used, each behavior changes how quickly the motivation to help rises. The three behaviors are created by modifying a single variable that controls this. The values tests are 0.5, 1.0, and 2.0. At the higher value, the motivation to help rises twice as fast as normal, and opposite for the lower value. This should give significant variety amongst the tests, from which to draw conclusions from. Which values are best for helping may depend on what the environment is like. On easier environments it may be best to have low help weight values, to prevent unnecessary merges. On harder environments, a high help weight can cause merges to occur sooner than normal, allowing the team of blobs to adapt faster.

4.1.4 Sensor Placement Threshold

One aspect in this project not seen in many areas of research, is mobile robots both exploring an unknown area and using that information to place survey sensors. The problems of multi-agent exploration, and optimal sensor placement have been studied extensively, but not nearly as much put together. In the current approach, a blob decides to place a sensor when the amount of unsurveyed area that has been explored rises above a certain threshold. It makes several approximations to do this, but it does work fairly well. By modifying this threshold higher or lower, the blob waits till more or less of

the map to be explored before placing a sensor. Three different values are tested, 25%, 50%, and 75%. The lower the value, the faster it is expected the blobs will place all their sensors and complete the simulation. The larger the value though, better accuracy is expected, as they have more information to consider when deciding sensor placement.

4.2 Simulation Metrics

Several metrics are measured each experiment. They determine how well the team of blobs completed their goals. A list of them is in Table 4.2.

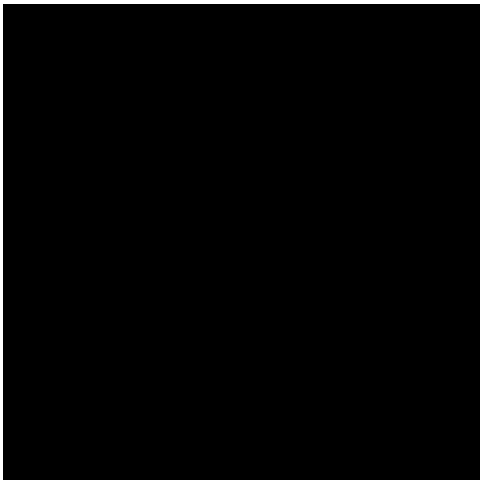
4.3 Comparison with Best Greedy Placement

Just running the simulation to see how well it can place sensors is not too useful if there is nothing to compare it to. In the experiments, the blobs are limited by what has been explored when they place sensors, and they also must use low resolution estimates of the viewsheds. Ideally, it would be best to compare those results to a known optimal placement for each map, but that is impractical. Instead, we calculate the best greedy placement if there is full knowledge of the map and the real viewsheds are used instead of estimates. A specialized program is used to calculate a minimum number of sensors that are required to survey at least 90% of the environments used. This is much more computationally expensive then a normal experiment, but it only needs to be run once per map. However many sensors are required to reach 90% is used in all experiments using that map. This way, the experiment results can see how they measure up to a baseline greedy placement, which was created with full knowledge of the map.

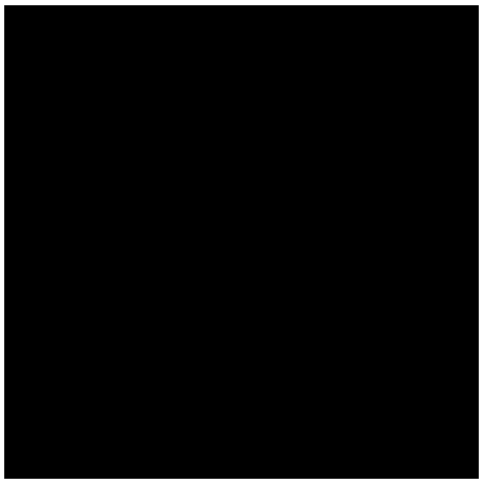
As said before, the survey problem is a variation of the classic Art Gallery problem. The idea is, given a Art Gallery, what is the minimum number and location of guards with line of sight vision needed to survey the gallery completely. It is a known NP complete problem, so when dealing with high resolution terrain like height maps, it is impossible to find the optimal in a reasonable time frame. Despite that, research has found some approximations that may be close to optimal. For example, it has been shown that using a

randomized greedy algorithm can get values near optimal in some cases. [5, 6] To find a good set of sensor locations, it uses the same greedy method the blobs use to find them, except with better data and more samples. In every 3x3 block in the map, one position is chosen and a viewshed is calculated for placing a sensor there. Then, of those viewsheds, the best one is chosen and a sensor is placed there. All remaining sample viewsheds are modified to remove areas that the sensor has surveyed The next sensor to place is chosen in the same manner. This goes on until the combined viewsheds of the picked locations is above a certain threshold. In this case, the threshold is 90% of visible cells. The number and placement of the sensors for each map can be seen in Figure 4.5. The number 90% was chosen, because it is a rather high percentage of the map, and a higher percentage would give less conclusive results. When 95% was used, the minimum number of sensors needed was much higher, almost double on some maps. Each extra sensor had diminishing returns, so it took many more to fill that last 5%. When blobs were given that many sensors they were often able to get very high results, because they had such an excess of sensors. With 90%, each sensor location is much more important, making the decision of where to place it have greater affect on the final results.

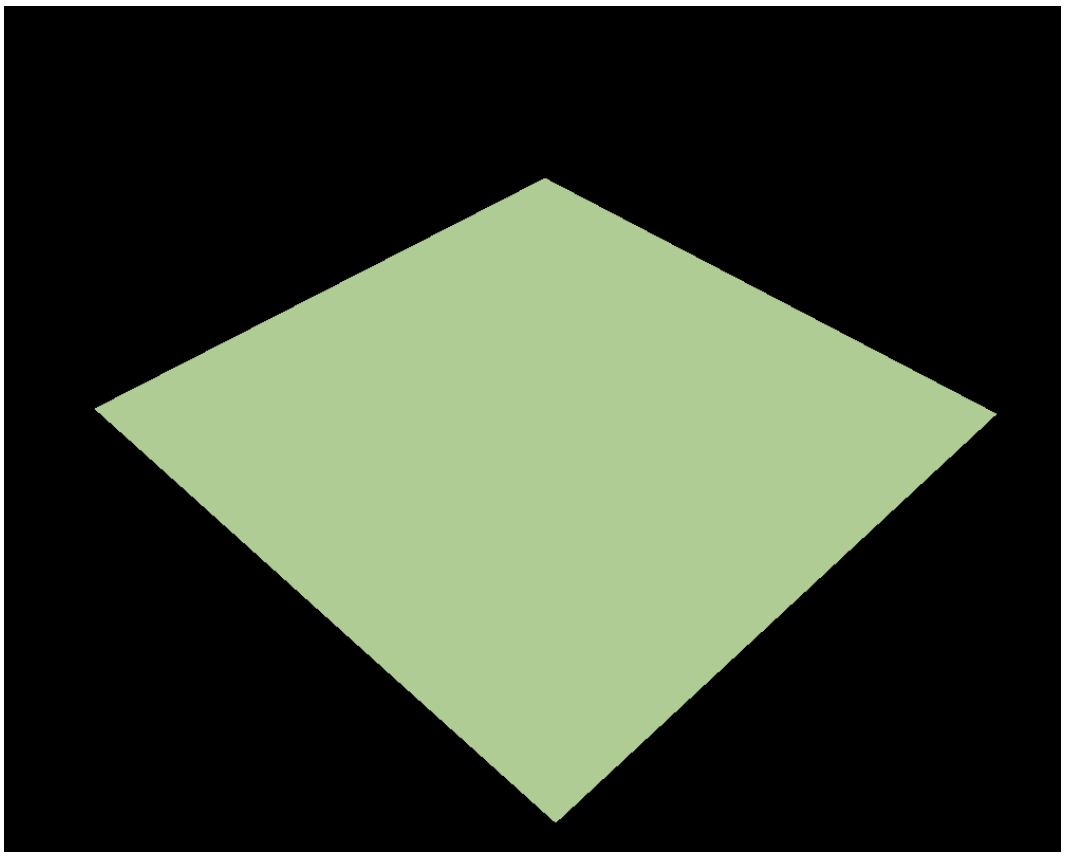
Each of the maps took 4 to 11 sensors to survey 90% of it. This is obviously not optimal, as can be seen on the flat map. It is possible to get 90% coverage of the flat map with just 3 sensors, but the greedy algorithm instead uses 4. In the flat map, many locations have the maximum possible coverage, and are equally desirable for choosing the first sensor. The Greedy algorithm chooses the very first one it comes across, which makes placing the other sensors not as efficient. The result of 4 sensors is still used in the experiments, because the blobs use the same greedy algorithm. In the other maps, none of them are optimal, but they do not have as obvious issues as the flat map does.



(a) Height Map

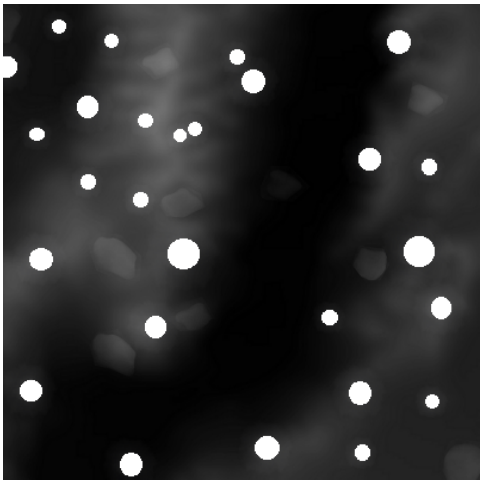


(b) Slope Map

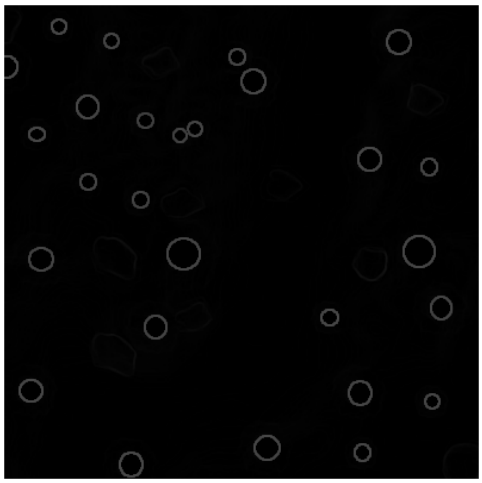


(c) 3D Map

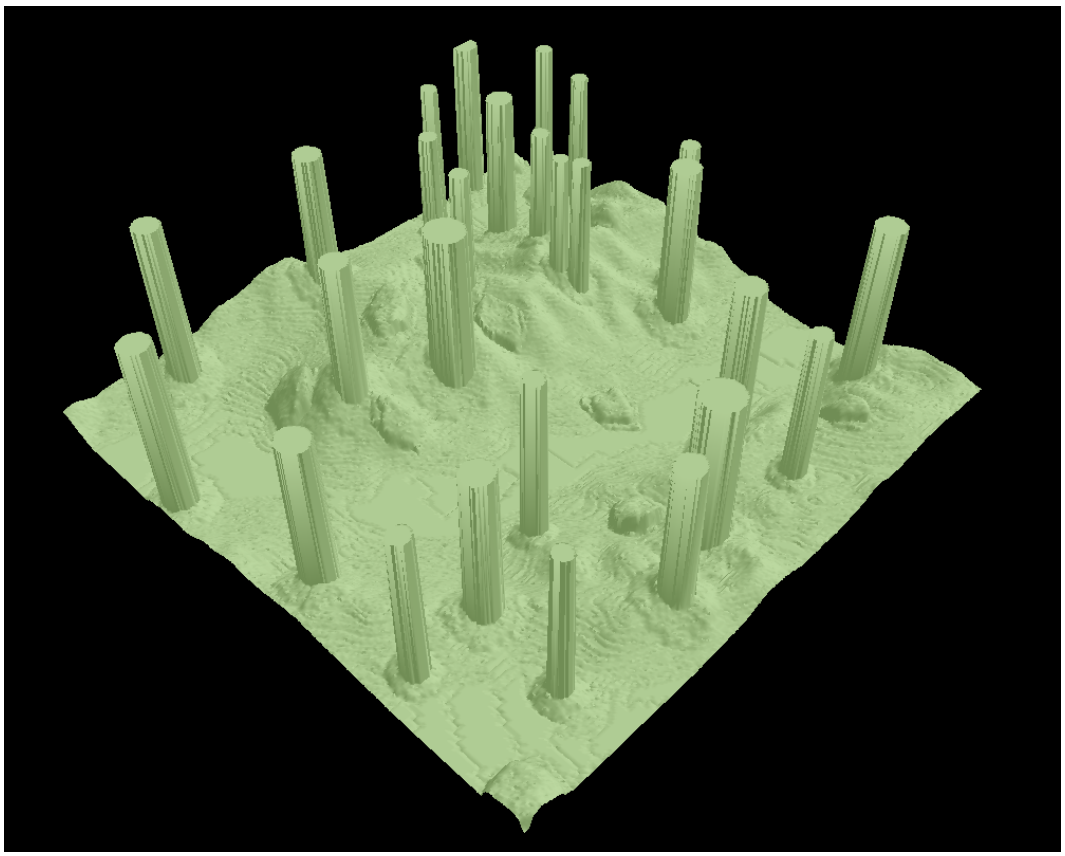
Figure 4.1: Experiment Map - Flat



(a) Height Map

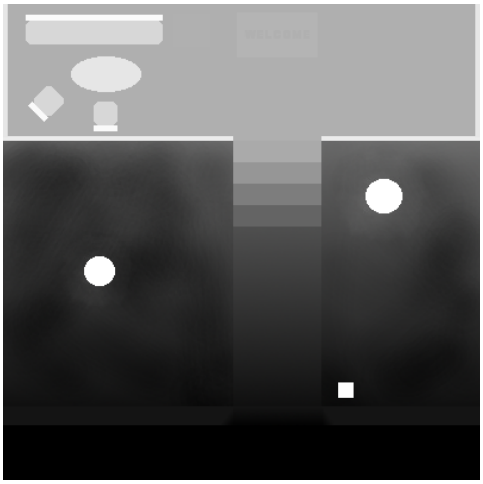


(b) Slope Map

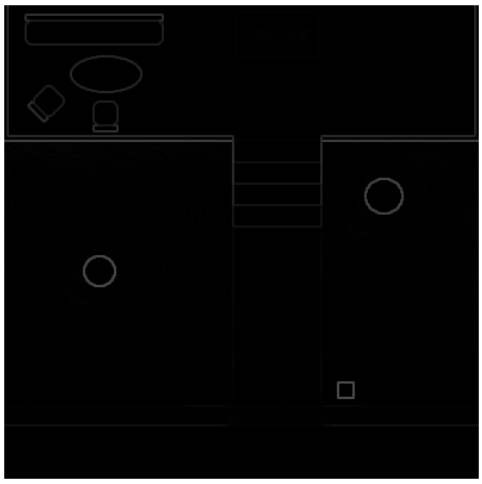


(c) 3D Map

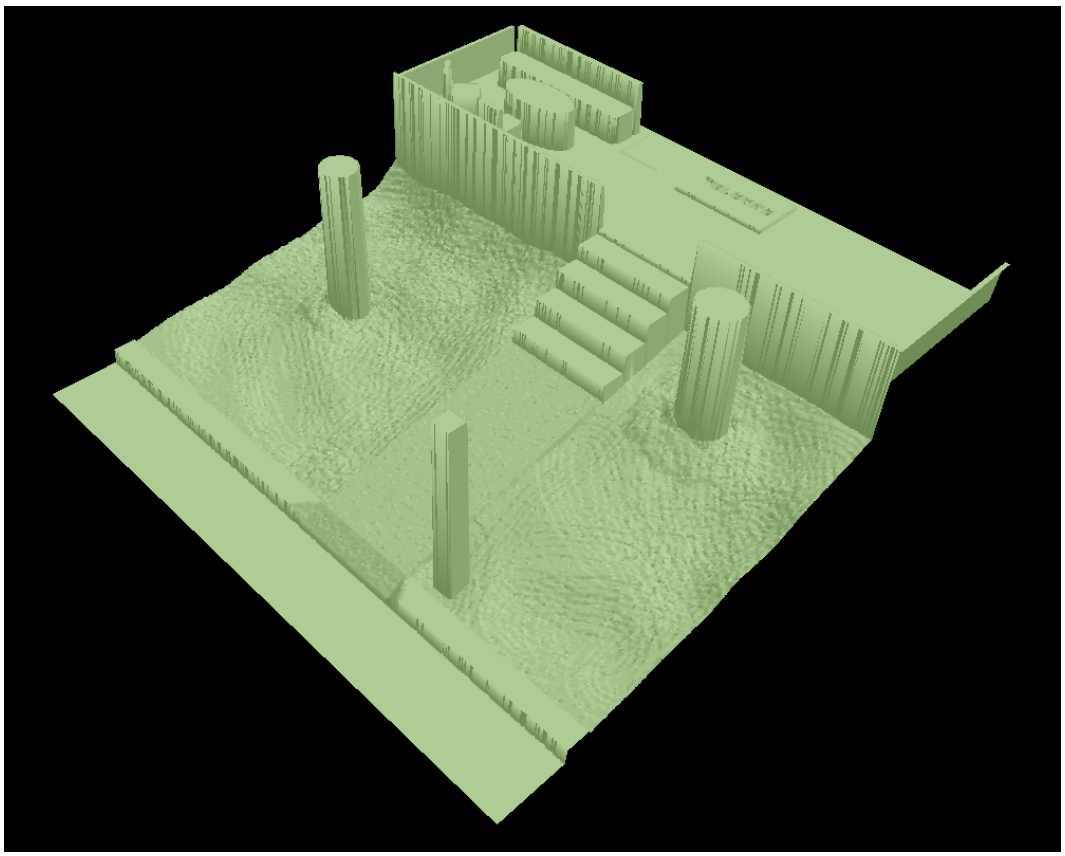
Figure 4.2: Experiment Map - Forest



(a) Height Map

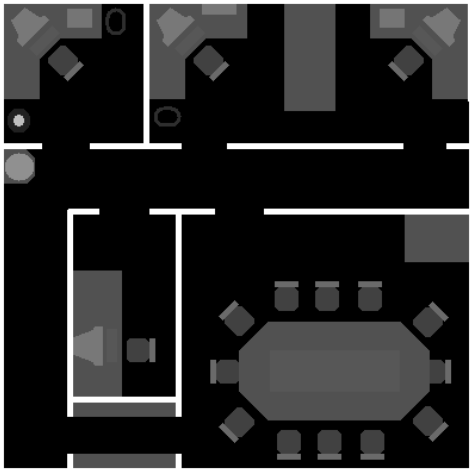


(b) Slope Map

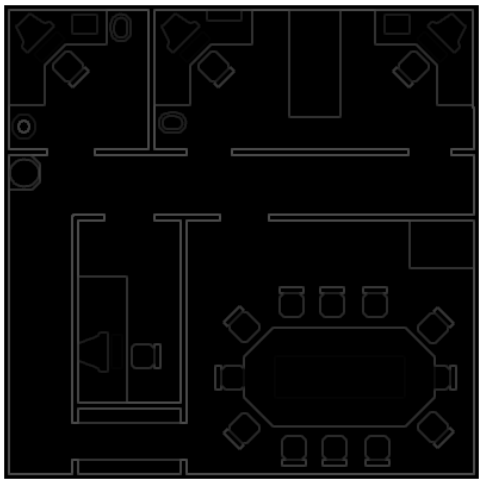


(c) 3D Map

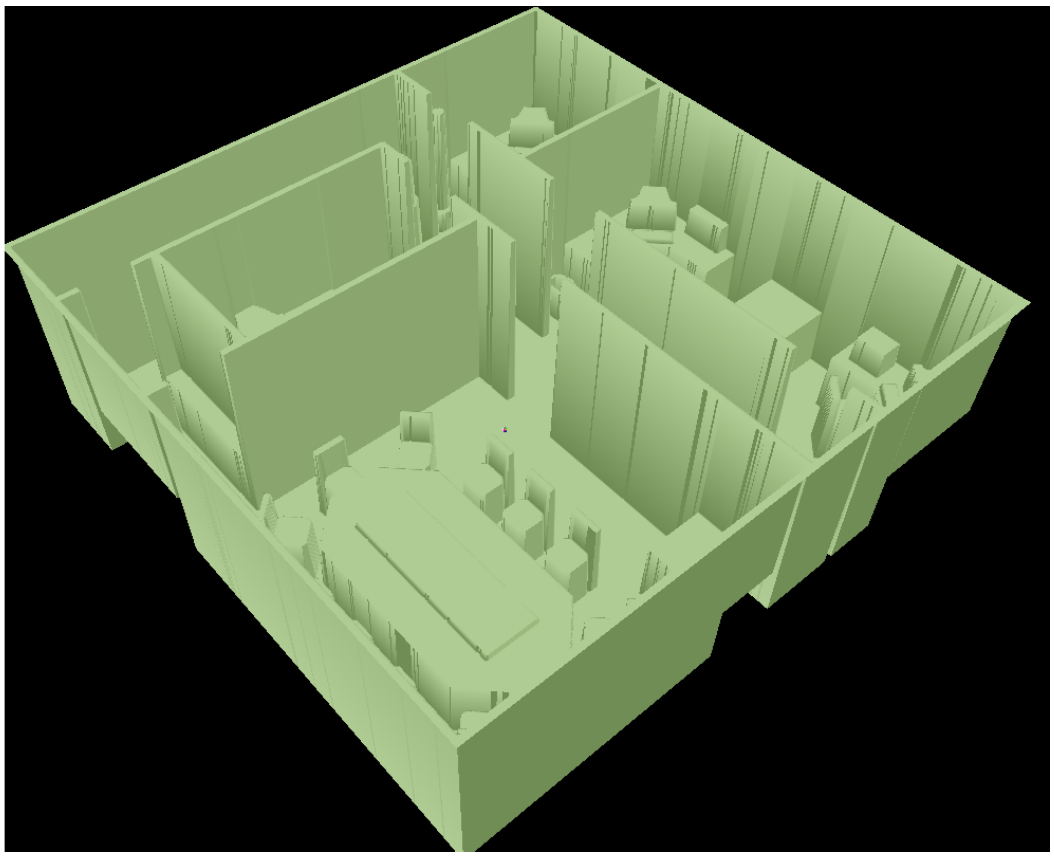
Figure 4.3: Experiment Map - Porch



(a) Height Map



(b) Slope Map

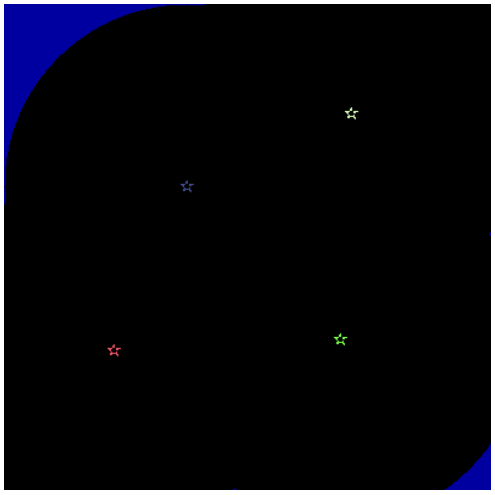


(c) 3D Map

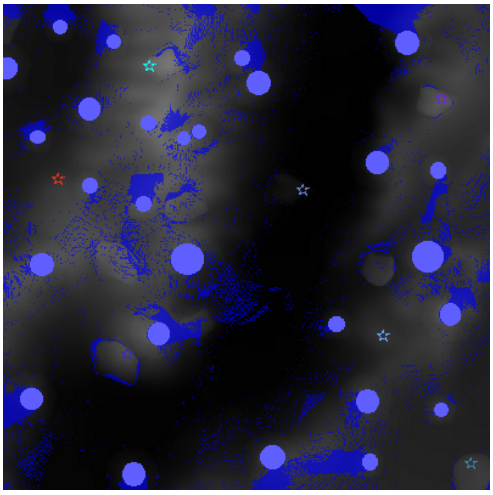
Figure 4.4: Experiment Map - Office

Number of Sensors Placed	Ideally, all sensors will be placed by the end of a experiment. Rarely, the simulation can time out due to a blob getting stuck, and not all are placed.
Total Steps	The total number of steps the simulation took is a better metric then how long the simulation took than real time, as it is not effected by processing speed or unavoidable thread scheduling issues.
Percent Surveyed	Percent of the map that was surveyed by the sensors it placed. The percentage leaves out unreachable areas like tree tops and walls. This metric shows how successful it was.
Percent Explored	Percent of the map that was explored during the simulation. It also leaves out unreachable areas. While not a direct metric for success, it is handy to use to compare with the survey percentage.
Number of Splits	Records the total number of times a blob had split.
Number of Merges	Records total number of times 2 blobs have merged together. Combined with number of splits, this is used to record the energy a blob used in modifying its configuration.
Weighted Displacement	Anytime a blob moves, the Weighted displacement is increased by the number of modules that are within that blob. This gives an estimate on how much energy has been used, as larger blobs will take more energy to move around then smaller ones. One blob of size 200 moving one cell adds as much to the weighted displacement as 4 blobs of size 50 each moving one cell.

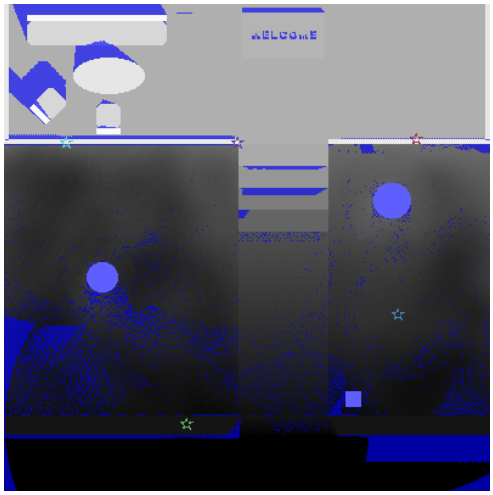
Table 4.2: Measured Simulation Metrics



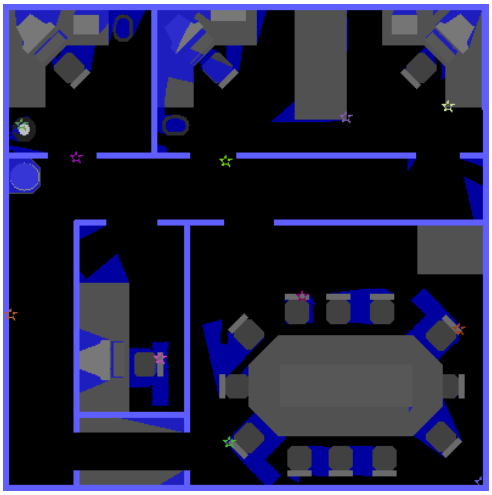
(a) Flat - 4 Sensors, 95.49% coverage



(b) Forest - 7 Sensors, 90.21% coverage



(c) Porch - 5 Sensors, 90.60% coverage



(d) Office - 11 Sensors, 90.71% coverage

Figure 4.5: Baseline Best Greedy sensor locations.

Chapter 5

Results and Discussion

5.1 Flat Map Results

The Flat map is the easiest map to traverse. It has no obstacles to vision or movement. Table 5.1 shows that Flat map simulations take the least amount of steps to finish in all experiments when compared to other maps in Tables 5.6, 5.16, and 5.11. These experiments only have 4 sensors to deploy, which are deployed in a greedy fashion. When it is in Limited mode, with no splits or merges, it takes much longer to complete its goals then when splitting is enabled. There is no solid trend when splitting is enabled but Help Weight is modified. This is because, as seen in Table 5.4, the Blobs primarily Split into many small blobs, and very rarely if at all merge back together. The map is easy enough that there are no obstacles to movement, so no reason to merge. As the Place Sensor Threshold is increased, the time to complete the experiment also rises. This is because it waits until more of the map is explored before placing sensors.

	Limited	**Help 0.5**	**Help 1.0**	**Help 2.0**
25%	2012.7 ± 370.8	813.4 ± 132.7	791 ± 116.9	812.5 ± 114.5
50%	2543.5 ± 271.4	967.2 ± 120.9	947.8 ± 87.6	1013.2 ± 113.8
75%	3399.1 ± 371.3	1114.3 ± 138.3	1099.1 ± 103.3	1091.6 ± 137.5

Table 5.1: Flat - Average number of steps taken

The best greedy sensor placement covers 95.49% of the Flat map (Figure 4.5a). The experiment results from simulation come quite close to that. They are all within 91.62 - 93.58%. While that is good overall, it does not give very strong trends for which is best. There is a slight trend upwards in the Limited Column, showing that the more it explores before placing a sensor,

the better its end results. But when splitting and merging are enabled, there is no such trend. In fact, the amount surveyed almost never changes. In the individual simulations where the Place Sensor threshold is 50% or above, a vast majority of them have surveyed the exact same amount, 92.21% of the map. The standard deviation is exactly zero for most of them. Every test run reached the same sensor placement in those cases. This case is shown in Figure 5.1. It is interesting that there is no improvement between a 50% and 75% threshold, except in the Limited case. There is a small amount of inherent randomness in the system, particularly where the blobs decide to explore when multiple locations have the same motivation. Even with that, the sensors are placed in the exact same locations a majority of the time.

	Limited	**Help 0.5**	**Help 1.0**	**Help 2.0**
25%	91.62 ± 0.99	92.07 ± 0.44	92.09 ± 0.41	91.66 ± 1.25
50%	92.08 ± 0.24	92.21 ± 1.82×10^{-5}	92.21 ± 0.0	92.21 ± 9.3×10^{-5}
75%	92.21 ± 1.82×10^{-5}	92.21 ± 0.0	92.21 ± 0.0	92.21 ± 0.0

Table 5.2: Flat - Average percent of reachable terrain surveyed

The Flat map is an easy map to explore, so it's not unexpected that a large amount of the map can be explored in a short time. The value of the Place Sensor threshold has a big effect on how much is explored. The lower it is, the less is explored. This trend is bigger than other maps, due only being 4 sensors to place. This is contrasted with the Survey amounts, which varied little when the Place Sensor threshold was changed. There is also a higher deviation in the results the lower it is, suggesting that when placing the sensors early, blobs without sensors have time to explore more area sometimes. Even though significantly different amount of terrain was explored, it seemed to come to the exact same sensor placement each time. The Help Weight on the other hand, has no noticeable effect. Since there is little to no obstacles, there is no need for blobs to help each other. But when both splits and merges are disabled in the limited, the amount of area explored is significantly less, due to there being no parallelism. In the cases with parallelism, when the last sensors are being placed by some of the blobs, the remaining ones can continue

exploring more of the map. By the end of these experiments, 98% of the map has been explored in most cases. The Flat map is easy enough, that no blob needs help from another.

	Limited	**Help 0.5**	**Help 1.0**	**Help 2.0**
25%	72.04 ± 9.85	85.39 ± 8.45	84.83 ± 8.75	85.05 ± 9.20
50%	81.75 ± 4.67	96.42 ± 2.59	95.86 ± 3.16	96.51 ± 3.56
75%	93.58 ± 3.44	98.69 ± 1.59	98.59 ± 1.36	98.58 ± 1.69

Table 5.3: Flat - Average percent of reachable terrain explored

In the all non-Limited cases, it is shown in the Flat map that the blobs almost never have to merge. In all cases they split at least 7 times, which is the maximum number of times a 1400 sized blob can split if the minimum split size is 300. The map is easy enough that they could split more, but the experiments don't last long enough for that to happen. The default minimum split size is set at 300, and it takes more time to change that. Only in the case where the Help Weight is much higher than normal, do the experiments have a chance of merging. It is rare because the only obstacles in the map are at its very edges. Only than might it attempt a merge, but the increase in splits as well shows that the merged blobs often split right apart soon after.

	Limited	**Help 0.5**	**Help 1.0**	**Help 2.0**
25%	0 / 0	7.0 / 0	7.0 / 0	7.05 / 0.2
50%	0 / 0	7.0 / 0	7.0 / 0	7.05 / 0.45
75%	0 / 0	7.0 / 0	7.0 / 0	7.1 / 0.3

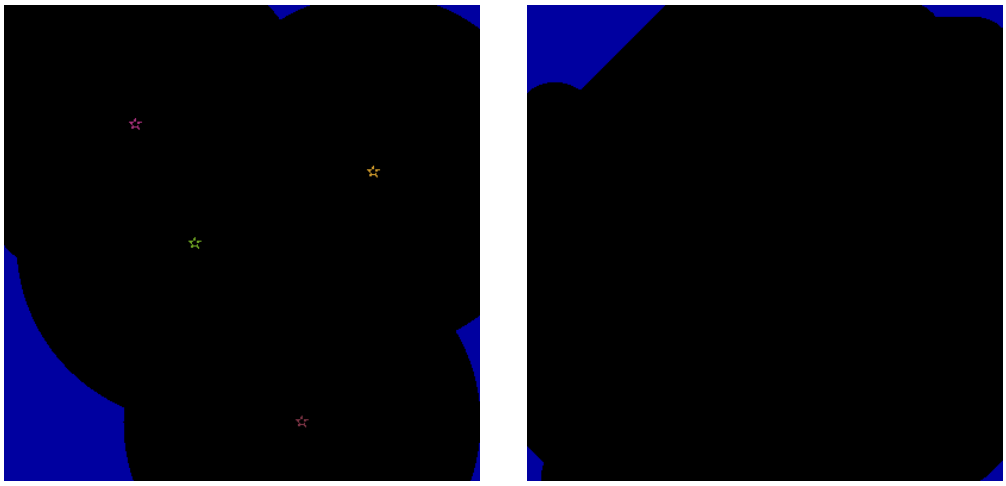
Table 5.4: Flat - Average splits/merges

The weighted displacement is an approximation of the overall energy used by the system to complete it goals. Compared to the Limited experiments, the experiments allowing reconfiguration use around 2.5 times less energy. This is because in the limited experiments, it has to drag the entire bulk of the blob around to each location. In a map as easy as this, two blobs half the size can accomplish almost twice as much using the same amount of energy. The value of the Help Weight does not have any effect on the energy

used, as in this simple map, there is little value in helping other blobs. The energy used does increase as the Place Sensor threshold is increased, and is due to more terrain being explored, and the simulations lasting longer.

	Limited	**Help 0.5**	**Help 1.0**	**Help 2.0**
25%	$3.206\times10^6 \pm 5.562\times10^5$	$1.289\times10^6 \pm 2.106\times10^5$	$1.252\times10^6 \pm 1.933\times10^5$	$1.272\times10^6 \pm 1.921\times10^5$
50%	$4.081\times10^6 \pm 4.611\times10^5$	$1.532\times10^6 \pm 1.911\times10^5$	$1.499\times10^6 \pm 1.400\times10^5$	$1.598\times10^6 \pm 1.733\times10^5$
75%	$5.465\times10^6 \pm 6.001\times10^5$	$1.759\times10^6 \pm 2.183\times10^5$	$1.730\times10^6 \pm 1.518\times10^5$	$1.724\times10^6 \pm 2.141\times10^5$

Table 5.5: Flat - Average total weighted displacement



(a) 92.21% of Map Surveyed (b) 95.79% of Map Explored

Figure 5.1: Experiment Result on Flat Map, 50% Sensor Threshold, 1.0 Help Weight. Showing final Survey sensor placement(a), and the final terrain explored. (b)

5.2 Forest Map Results

The Forest map is quite a bit more complicated than the Flat map, but is still very easy to traverse. There are many trees that both block vision and movement. The Forest Map simulation time follows similar trends as the Flat map. The experiments take much more time when they are limited from splitting. Time to finish also increases as the Place Sensor threshold is increased. The difference is that the value of the Help Weight does seem to make a small difference. As the Help Weight increases, the steps taken increases slightly. This means blobs are spending some time helping each other, instead of none at all. In the Flat map, there was little to no change, as the blobs had no need to merge together to help. While the Forest map is still simple to traverse, merges occasionally still occur, as shown in Table 5.9.

	Limited	**Help 0.5**	**Help 1.0**	**Help 2.0**
25%	2959 ± 167.1	951.4 ± 133.0	987.4 ± 186.6	1109.8 ± 191.1
50%	3972.2 ± 334.7	1166.7 ± 129.7	1168.3 ± 109.9	1345.3 ± 183.4
75%	4712.2 ± 291.9	1178.7 ± 111.3	1292.8 ± 150.1	1419.4 ± 287.7

Table 5.6: Forest - Average number of steps taken

When the amount percentage of the terrain surveyed is calculated, it does not include the areas that are completely unreachable. The best greedy placement covers 90.21% of the reachable areas of the map using 7 independent sensors. The best results from the experiments were almost 87% of the map, so it was pretty close. Interestingly, there is no significant difference of surveyed area when the Help Weight is changed, or even splits and merges disabled. The map is easy enough to traverse, that having extra help or no help is inconsequential to getting good sensor locations. Since higher Help weight values do increase the steps taken as seen in Table 5.6, it does not seem beneficial to have a high Help Weight for this map. There is significant difference in percent surveyed based on the Place Sensor threshold. A higher threshold increases the percent surveyed by quite a bit, along with an increase in time taken. This is due to it waiting longer before placing sensors, which ends up being in better locations overall.

	Limited	Help 0.5	Help 1.0	Help 2.0
25%	77.35 ± 3.45	78.53 ± 4.30	78.07 ± 2.72	77.78 ± 3.27
50%	83.46 ± 2.37	84.29 ± 2.17	85.66 ± 1.69	84.11 ± 2.51
75%	85.89 ± 1.56	86.69 ± 1.72	86.93 ± 2.09	85.79 ± 1.89

Table 5.7: Forest - Average Percent of reachable terrain Surveyed

Because most of the non-tree terrain is easy to traverse and see, Blobs can explore the vast majority of the map in a rather short period of time. In Limited Mode, the blobs explore significantly less of the map before they have placed all their sensors, but otherwise, the Help weight does not affect it. The Place Sensor threshold on the other hand has a direct relationship with area explored. With a threshold of 75%, the blobs come close to exploring 100% of the terrain before the experiment ends. Like the Flat map, there is much more variation in how much is explored when the Place Sensor threshold is low. It sometimes explores a higher ratio of terrain than it did on the simpler Flat map. This is probably due to there being more sensors to place, and as such a higher period of time to explore.

	Limited	Help 0.5	Help 1.0	Help 2.0
25%	77.75 ± 2.60	91.45 ± 6.00	90.67 ± 6.03	91.06 ± 6.53
50%	90.04 ± 2.13	97.67 ±2.34	97.98 ± 1.75	97.75 ± 1.93
75%	95.48 ± 1.48	98.73 ± 1.07	99.27 ± 1.06	98.85 ± 1.02

Table 5.8: Forest - Average Percent of reachable terrain Explored

Since the Forest map is relatively easy, not many merges are needed to be successful. When the Help Weight is low, it tends to split until it goes under the minimum size at 300, and rarely merge. When the Help Weight is 2.0, it does merge more some, but it does is not seem to be beneficial. There is no increase in terrain explored or surveyed because of it, just an increase in steps taken and energy used.

There is a huge difference in overall energy used when comparing the Limited and non-Limited modes. The non-Limited experiments have 3-4 times less weighted displacement. In this map, it is very beneficial to split up and

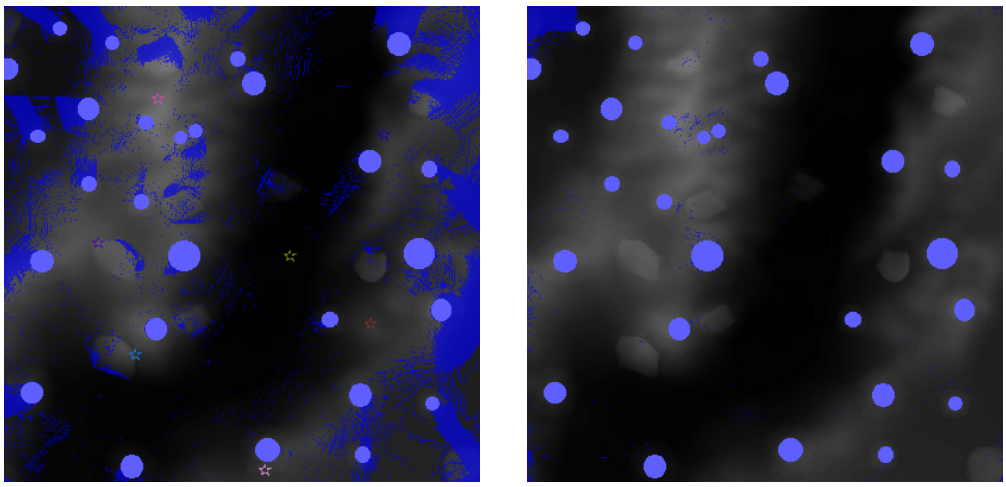
	Limited	**Help 0.5**	**Help 1.0**	**Help 2.0**
25%	0 / 0	7.0 / 0.05	7.05 / 0.25	7.4 / 2.25
50%	0 / 0	7.0 / 0.1	7.1 / 0.7	7.58 / 2.64
75%	0 / 0	7.0 / 0.15	7.0 / 0.55	7.8 / 3.05

Table 5.9: Forest - Average splits/merges

parallelize multiple tasks than attempt them one at a time. There are also slight increases in energy used as the Place Sensor threshold is increased. The correlated increases in survey coverage are shown in Table 5.7 make it a worthwhile trade. Conversely, the effect of increasing the Help Weight is not worthwhile. It increases the energy used, with no positive impact on survey coverage.

	Limited	**Help 0.5**	**Help 1.0**	**Help 2.0**
25%	$4.711\times10^6 \pm 2.677\times10^5$	$1.500\times10^6 \pm 2.061\times10^5$	$1.559\times10^6 \pm 2.912\times10^5$	$1.709\times10^6 \pm 3.029\times10^5$
50%	$6.375\times10^6 \pm 5.308\times10^5$	$1.835\times10^6 \pm 2.013\times10^5$	$1.844\times10^6 \pm 1.648\times10^5$	$2.112\times10^6 \pm 2.914\times10^5$
75%	$7.477\times10^6 \pm 5.016\times10^5$	$1.871\times10^6 \pm 1.729\times10^5$	$2.053\times10^6 \pm 2.485\times10^5$	$2.246\times10^6 \pm 4.395\times10^5$

Table 5.10: Forest - Average total weighted displacement



(a) 84.28% of Map Surveyed (b) 98.84% of Map Explored

Figure 5.2: Experiment Result on Forest Map, 50% Sensor Threshold, 1.0 Help Weight. Showing Survey sensor placement(a), and the total terrain explored (b)

5.3 Porch Map Results

Most of the Porch map is mostly open spaces which are easy to traverse, and not much of it is completely unreachable. Only the railings of the Porch need the entire Blob together to be climbed upon. As with other maps, the Limited mode takes significantly more steps than non-Limited modes. Without any cooperation it can't explore the large amount of easy terrain quickly. Also common to the other maps, the higher the Place Sensor threshold is, the longer the experiment takes. There isn't a solid trend when the Help Weight is changed. Because of the difficult areas in the Porch map, blobs merge together much more often than in the Forest and Flat maps, as seen in Table 5.14. There is more deviation in the steps taken in the experiments than compared to the Flat and Forest maps. This probably has to do with the same reason the survey results vary so much.

	Limited	**Help 0.5**	**Help 1.0**	**Help 2.0**
25%	3085.4 ± 593.1	1628.8 ± 642.2	1350.1 ± 436.4	1359.6 ± 290.3
50%	4038.8 ± 504.2	1840.8 ± 606.4	1824.9 ± 509.2	1765.1 ± 413.3
75%	4866.3 ± 428.6	2635.9 ± 529.7	2285.6 ± 449.6	2329.9 ± 517.6

Table 5.11: Porch - Average number of steps taken

The survey results on the Porch map are interesting and very different than those on the Forest and Flat map. It was found that the best greedy placement of 5 sensors in Figure 4.5c covered 90.60% of the map. This placement depends on having 3 of the 5 sensors on the porch railing, which is a difficult area to reach. If it misses placing sensors up there, end results are affected heavily. We see that as the highest survey percentage is only 85.73%, and it is in the Limited column with the threshold at 75%. In the cases where splitting is allowed, results are significantly worse. It seems that because the initial blob never splits, it is able to explore the difficult areas before it places sensors. While the trend when the Place Sensor threshold is increased is ordinary, the Help Weight trend is far from it. When the threshold is 25%, the amount surveyed actually decreases as the Help Weight is increased. It's almost flat at 50% and 75%. This likely occurs because when the Sensor Placement threshold is low, it doesn't have time to explore the small railing

areas that are the best sensor locations. The majority of its placements are in easy locations, which don't require any other blob help. When the Sensor Placement threshold is high, it is more likely to have explored the best sensor locations, so it requires help to get to them. In the example in Figure 5.3, The upper areas were explored last, so it did not place any sensors on the railings, which were the best areas. But in Figure 5.4, where the no splitting was allowed, it was able to explore most of the upper porch and place 2 of its 5 sensors in good locations on the railing. It is only possible to climb onto the railings from the top of the porch. To reach the area, it must climb up the steps. It looks like in most of the simulations that allow splitting, the blobs explored the easier lower portion first, and only got to the upper portion after it had placed many of its sensors.

	Limited	**Help 0.5**	**Help 1.0**	**Help 2.0**
25%	75.83 ± 4.56	75.15 ± 5.01	72.45 ± 4.03	71.15 ± 5.39
50%	82.51 ± 3.40	79.23 ± 3.57	77.49 ± 4.34	78.05 ± 3.87
75%	85.73 ± 2.79	80.08 ± 3.21	80.89 ± 3.39	80.75 ± 3.13

Table 5.12: Porch - Average percent of reachable terrain surveyed

While the explored ratios increase as the Place Sensor threshold is increased, there isn't a clear trend as the Help Weight is changed. Oddly, the highest explored percentages are when the Help Weight is only 0.5. High Survey results don't correspond with high explore percentages either. The good locations for sensors are difficult to get to, but are a very small percentage of total terrain. It's likely that when the help percentage was lower, it explored more of the easy terrain and ignored the harder terrain. It was able to explore more overall, but because it missed the good sensor locations, the survey results were not as good.

	Limited	**Help 0.5**	**Help 1.0**	**Help 2.0**
25%	74.74 ± 4.47	88.96 ± 7.85	84.20 ± 6.12	81.03 ± 6.76
50%	88.23 ± 1.90	93.89 ± 4.75	92.72 ± 3.92	91.92 ± 3.51
75%	95.04 ± 1.13	98.17 ± 1.58	93.03 ± 1.72	96.63 ± 1.82

Table 5.13: Porch - Average percent of reachable terrain explored

The rate of merging is much higher on the Porch map then with the Flat and Forest map. The Split rate is also higher, meaning that after blobs have merged, they often split apart again. There is no guarantee that it completed the difficult task it merged for before it splits up again. It is shown that as the Place sensor threshold raises the number of splits and merges also rises. The number of merges gets closer to the number of splits as well. This is likely partially because the length of the simulation is increased, meaning there are more chances to split and merge. But the increased merges relative to splits could be because the blobs are exploring the rare difficult terrain, so they need more help doing so.

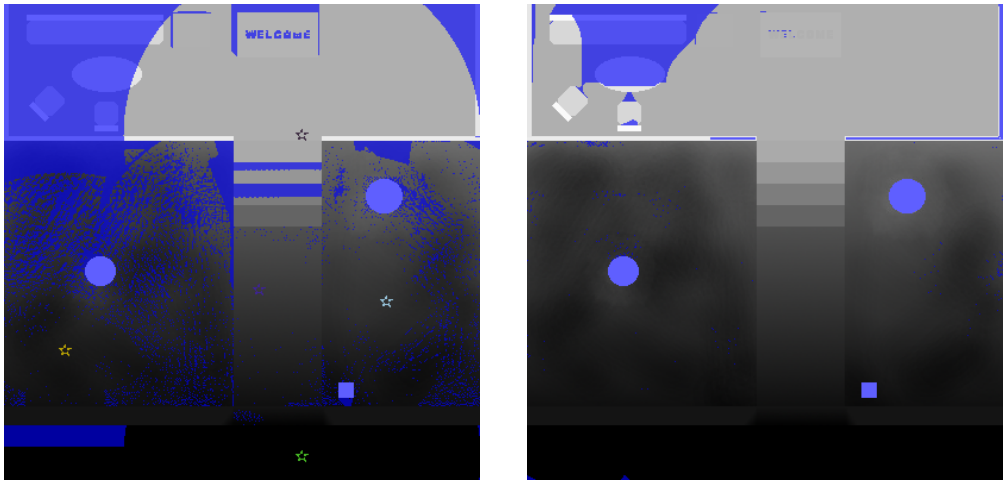
	Limited	**Help 0.5**	**Help 1.0**	**Help 2.0**
25%	0 / 0	8.85 / 5.23	8.3 / 4.6	8.6 / 5.55
50%	0 / 0	8.81 / 6.04	9.05 / 6.6	9.25 / 7.25
75%	0 / 0	10.05 / 8.95	10.15 / 9.15	9.9 / 8.7

Table 5.14: Porch - Average splits/merges

Its a common trend for all maps, that the weighted displacement increases as the Place Sensor threshold is increased. The numbers for the Porch map are higher than those of the Flat and Forest map, but that is expected as it is a more difficult map. The Limited column is also much higher than non-Limited columns like the others, but to less extent. This is likely because unlike the Flat and Forest maps, there is a significant amount of merging going on, so there is less parallelism overall. Modifying the Help Weight does cause some changes, but without any obvious trend.

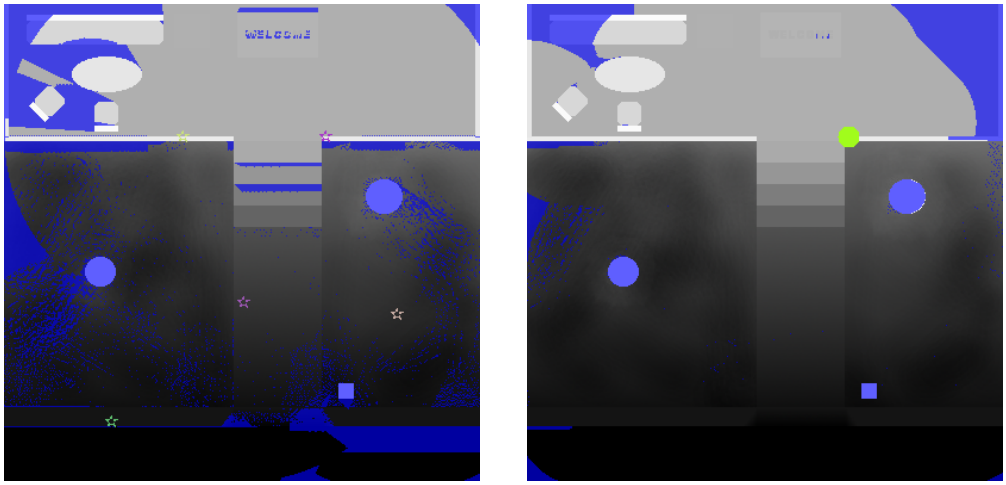
	Limited	**Help 0.5**	**Help 1.0**	**Help 2.0**
25%	$4.812\times10^6 \pm 8.949\times10^5$	$2.313\times10^6 \pm 7.629\times10^5$	$2.031\times10^6 \pm 5.901\times10^5$	$2.084\times10^6 \pm 4.061\times10^5$
50%	$6.375\times10^6 \pm 7.884\times10^5$	$2.727\times10^6 \pm 9.081\times10^5$	$2.660\times10^6 \pm 6.416\times10^5$	$2.700\times10^6 \pm 6.41\times10^5$
75%	$7.755\times10^6 \pm 6.853\times10^5$	$3.904\times10^6 \pm 8.294\times10^5$	$3.471\times10^6 \pm 7.237\times10^5$	$3.672\times10^6 \pm 8.791\times10^5$

Table 5.15: Porch - Average total weighted displacement



(a) 77.41% of Map Surveyed (b) 95.17% of Map Explored

Figure 5.3: Experiment Result on Porch Map, 75% Sensor Threshold, 1.0 Help Weight. Showing Survey sensor placement(a), and the terrain explored (b)



(a) 77.41% of Map Surveyed (b) 95.17% of Map Explored

Figure 5.4: Experiment Result on Porch Map, 75% Sensor Threshold, Limited. Showing Survey sensor placement(a), and the terrain explored (b)

5.4 Office Map Results

The most difficult map of the four is the Office one. It has many steep edges which require almost all 1400 modules to be in one blob to traverse. The Office map takes the longest to complete compared to all the other cases. It has the most sensors to place, and an environment with many obstacles to vision. In the best greedy placement in Figure 4.5d, it takes 11 sensors to cover 90.71% of the map. A little more than half of the sensors are placed at high locations like desks or chairs. The rest are on flat areas with large line of sight, like hallways or doorways. Compared to the other maps, Office map experiments do not have as much difference between the Limited and non-Limited cases. In Table 5.16, the number of steps taken just varies a small amount between the two. This is because, as seen in figure 3.8, the map is difficult enough to where the minimum split size rises above 1400. The Blobs learned to put themselves in a limited mode after some time. The initial segment where they are not limited does help explore parts of the map sooner, giving slightly better times overall. Increasing the Help weight also slightly increases the time the simulation takes. Assisting other blobs instead of placing sensors early does take extra time.

	Limited	**Help 0.5**	**Help 1.0**	**Help 2.0**
25%	6365.5 ± 856.9	5019.1 ± 1255	5030.7 ± 991.2	5743.4 ± 1441
50%	7276.7 ± 563.4	5392.0 ± 1183	6351.1 ± 1152	6469.4 ± 1048
75%	7374.4 ± 794.0	6849.8 ± 714.4	6984.9 ± 745.8	7058.7 ± 814.1

Table 5.16: Office - Average number of steps taken

What stands out in the Survey results for the Office map, is that the Limited method has the best results. Its results are consistently around 88%, even when the Place Sensor threshold is low. When splitting is enabled, the percentage increases as the Place Sensor threshold increases like normal, but only reaches as high as Limited did at 75%. Since limited has the best results, it must mean that the blobs often miss the harder but better sensor locations when they are split apart. As seen in Figure 3.9, many sensors had already been placed before the Blob's minimum split size had reached the maximum. The easiest locations were explored first, so that is where it started placing

sensors. In Figure 5.5b, we see that more of the sensors are placed on the ground floor then in the best greedy in Figure 4.5d. When the blob is all together, it treats the climb from the floor to the top of a desk the same as traversing the same horizontal distance across the floor. It is equally likely to explore difficult and easy locations, unlike blobs of smaller size. Like all other maps, it gets better results the higher the Place Sensor threshold is, as the cost of longer execution time and higher energy usage.

	Limited	**Help 0.5**	**Help 1.0**	**Help 2.0**
25%	87.36 ± 3.63	82.05 ± 4.14	83.24 ± 3.54	83.48 ± 4.18
50%	88.88 ± 1.49	85.19 ± 1.94	86.92 ± 2.37	86.68 ± 2.81
75%	88.72 ± 1.42	88.63 ± 1.15	88.65 ± 1.24	87.90 ± 1.113

Table 5.17: Office - Average Percent of reachable terrain surveyed

Unique to the Office map, the highest percentage of the map explored occurs when it is limited to no splits or merges. Even in the Porch map, which had similar problems with sensor placement, the system explored more when splitting and cooperation was enabled. It keeps the predictable trend as the Place Sensor threshold is increased, but not with the Help weight. The results are highest with the weight at 1.0, with both 0.5 and 2.0 being lower. It's not a expected result, or easy to find the cause of. Higher help weight should cause the minimum split size to reach the maximum faster, locking the blob into the seemingly superior Limited mode. It could come into play near the end of the simulation, after the location of the last sensors has been decided. If it's placing a sensor at an easy or medium difficulty location, the blob might split again, one to place the sensor, and the other to explore new ground. A very high Help weight might encourage them to merge again before there was much extra exploring.

	Limited	**Help 0.5**	**Help 1.0**	**Help 2.0**
25%	95.69 ± 5.70	88.64 ± 5.52	91.09 ± 3.78	90.33 ± 5.26
50%	98.98 ± 2.03	91.41 ± 2.82	95.35 ± 3.16	94.35 ± 3.07
75%	98.13 ± 2.27	96.64 ± 1.74	97.21 ± 1.96	96.05 ± 2.15

Table 5.18: Office - Average Percent of reachable terrain explored

During all simulations on the Office map, the number of splits and merges are very close. Because of the difficulty of the map, almost every time it splits, it will have merged back together by the end of the simulation. The minimum split size rises above 1400 almost universally in experiments where splitting and merging are enabled. The numbers of splits and merges increases slightly as the Place Sensor threshold does, but not enough to suggest it's not solely due to the increased simulation length.

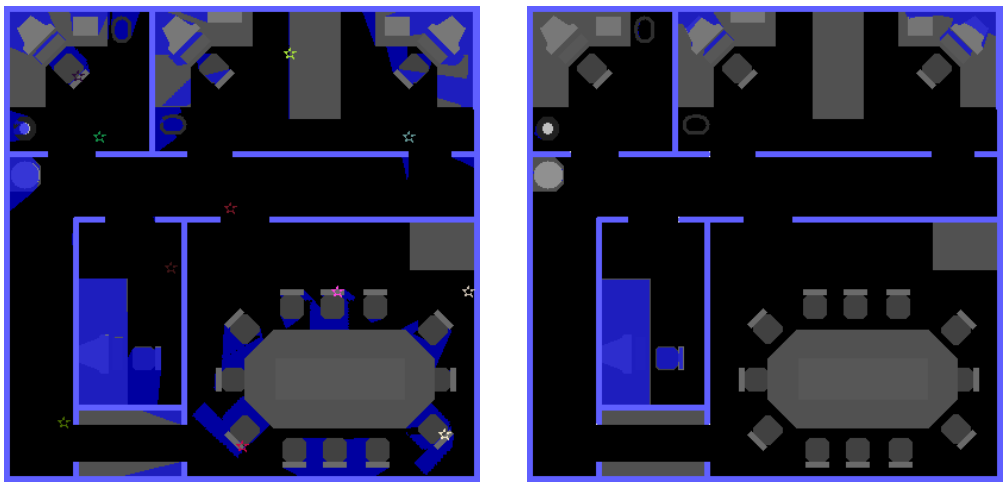
	Limited	**Help 0.5**	**Help 1.0**	**Help 2.0**
25%	0 / 0	9.85 / 9.6	9.50 / 9.40	9.55 / 9.4
50%	0 / 0	9.95 / 9.75	9.85 / 9.60	9.66 / 9.57
75%	0 / 0	10.6 / 10.3	9.80 / 9.80	10.19 / 10

Table 5.19: Office - Average splits/merges

Of all maps tested, experiments on the Office map take the most energy to accomplish their goals by far. Because a significant amount of the map cannot be explored without the whole blob being together, the Limited column is not much larger then respective values with splitting. There is a slight advantage for energy used when the Place Sensor threshold is smaller, but the advantage disappears as it gets larger. There isn't a general trend as the Help Weight is changed. Overall, the experiments on the Office map are much more similar with each other than experiments on all the other maps.

	Limited	**Help 0.5**	**Help 1.0**	**Help 2.0**
25%	$1.035\times10^7 \pm 1.459\times10^6$	$7.987\times10^6 \pm 2.897\times10^6$	$8.174\times10^6 \pm 1.841\times10^6$	$9.640\times10^6 \pm 3.725\times10^6$
50%	$1.187\times10^7 \pm 7.076\times10^5$	$8.791\times10^6 \pm 2.733\times10^6$	$1.088\times10^7 \pm 2.985\times10^6$	$1.037\times10^7 \pm 1.691\times10^6$
75%	$1.197\times10^7 \pm 1.135\times10^6$	$1.173\times10^7 \pm 2.654\times10^6$	$1.129\times10^7 \pm 1.623\times10^6$	$1.132\times10^7 \pm 1.482\times10^6$

Table 5.20: Office - Average total weighted displacement



(a) 84.28% of Map Surveyed (b) 98.84% of Map Explored

Figure 5.5: Experiment Result on Office Map, 50% Sensor Threshold, 1.0 Help Weight. Showing Survey sensor placement(a), and the total terrain explored (b).

Chapter 6

Conclusion and Future Work

The results from the experiments show that Self Reconfiguring Robots can successfully complete a series of ordered complex tasks. On all given maps, they are able to successfully explore almost all of the reachable terrain, even when it's quite difficult to reach. They are also able to deploy a limited number of survey sensors in a way which is close to the best greedy solution. In all the experiments run, the blobs were able to complete the task of deploying all its sensors in a reasonable amount of time. It always placed sensors in useful locations, and often in very good locations. The sensors were never grouped together, and each sensor covered terrain no other sensor did.

In all simulations, the modules started as a single blob. But in the cases where self reconfiguration was enabled, it always took less time and energy to complete the goals. In most cases significantly so. Enabling reconfiguration also had a significant positive effect on some maps. On the easier maps, allowing reconfiguration increased the explored ratio considerably, and the surveyed ratio moderately. Parallel exploration allowed more terrain to be surveyed in much less time, meaning better sensor locations were drawn from that. Because there weren't many obstacles, it could reach these positions easily. On harder maps, allowing reconfiguration sometimes had a positive effect on the exploration ratio, but sometimes not. The effect on survey ratio was more consistent, with it being a small amount lower than when blobs where limited from reconfiguration. When blobs are exploring a terrain, they focus on exploring the easy terrain, but in harder maps, the difficult terrain usually contains the best survey locations. They miss these locations early on, causing worse survey results.

To better understand aspects of the blob's cooperation, two variables were experimentally varied. The first was the threshold of terrain that needed

to be explored but not surveyed before it would consider placing a sensor. Increasing this caused the Blobs to explore more area before placing each sensor. Exploration and surveillance are inter-related tasks in this case. The threshold controls the rate at which the blobs progress from one task to the other. When changing this variable, the affect was predictable and consistent. When it was increased, all measured values increased as well, including simulation time taken, percent surveyed and explored, number of splits and merges, and total energy used. Generally speaking, the higher the threshold, the better it can survey the area, but the higher its costs were in time and energy.

The second variable affects the willingness of blobs to help one another. It was a constant that directly weighted any increase in motivation for the Help task. Its motivation rose because of difficult terrain, and other blobs asking for help. When lower, blobs would be slower to come around to helping each other, and the opposite is true for increasing it. Changing this variable was not as predictable or consistent as the Place Sensor threshold. Some trends remained consistent; increasing the Help Weight increased the number of merges, splits, time taken, and energy used. On the other hand, the measurements of success, the explore and survey percentages, were not so consistent. On easier maps, changing their willingness to help had little to no affect on the results. This is due to few merges being needed to complete the goals. On harder maps, it was much more unpredictable, and it's harder to draw solid conclusions from. Higher values helped in some conditions on some maps, but in others it was harmful.

6.1 Future Work

These simulations took many people's work in new directions, and laid a base where many things can be expanded from. It in some way, it created more questions then found answers. The field of Self Reconfigurable Robots is still in its beginning stages, and can benefit greatly from further simulation work. Just because there is many improvements that can be done, does not mean that the results gained already are without value. The conclusions gathered from them guide the focus of future discoveries.

6.1.1 Simulation Improvements

Because this is a simulation, many assumptions are made to simplify it. Most of the assumptions are based on real systems, but some could still be replaced by more complex and better models. One in particular is the movement model for the Blobs. The model used has several simplifications. One in particular was a failure to account for the cost of movement in difficult terrain. While it kept track of the number of modules traversing a cell, it did not matter if that cell was steep or flat, just if it was traversable or not. It took the same amount of energy and time to traverse such a cell. A improved model may not just estimate the extra energy used climbing a slope, but also estimate how much extra time it might take. It's not very reasonable for a Blob to traverse a flat cell and a cliff in the same amount of time. The same goes for assuming a blob can split or merge in that time.

A second improvement to the movement model would be to use values measured from a real Self Reconfiguring Robot system. The sensor model could also use measured values, but it would be in the movement model that the greatest changes would be seen. Instead of basing traversability purely on slope, a real SRR system may move quite differently. It might depend on specific features of the map, or have different methods for climbing up or down the same slope. With a better estimate of energy used while movement, more accurate conclusions can be drawn. Tied with the above changes, you could be done to test that particular robot system while varying the same control values. Results could then be compared to results gathered in non-simulated environments with that same system

6.1.2 Sensor Placement

One issue was due to the greedy method for placing sensors. When a greedy method is used with full knowledge of the map, it gives pretty good results, as seen in Figure 4.5. It chooses the best location for a sensor at each step, since everything is explored. The issue comes up when only partial knowledge of the map is available. In many cases, it's likely the best location is not explored, so it won't consider placing a sensor there. When sensors are not placed at the most desirable location, remaining viewsheds are updated,

and the best locations utility could become much less. The prime example of this is the results on the Porch map in Table 5.12. The blobs focused on exploring the easy terrain, and place their first sensors in bad locations. By the time it explores the best locations, the utility of a sensor there is either much lower then it could be, or there are no sensors left to place. This greatly affects the final results. Greedy based algorithms depend greatly on the order results are chosen, and if early choices are short sighted or bad, it doesn't redo an earlier selection to recover.

This issue could be resolved better in two ways. First, the way a blob explores a map could be changed. Instead of valuing easy terrain over harder terrain, it could do the opposite. In the current system, if an area turns out to be unreachable, it will lower the motivation for that area slightly to see if there are easier areas to go for. Only after it runs into several areas that are difficult will it get help. After a merge happens, the motivation for the harder areas might not have recovered, so it might still avoid the area. While is should still lower the motivation for areas it can't reach temporarily, the motivation should recover, and possibly go higher than normal when a merge takes place, that way it is encouraged to go back and try to get to the hard areas it merged for.

Second, an approach other than the current greedy one could be used. It would be more complex, and might require moving sensors that have been placed. A simple improvement would be to still use greedy placement, but allow the sensors to be moved again once they were placed. Instead of considering already placed sensors static, it would recalculate the greedy locations for all currently placed sensors, plus one more. If it found that there was significant benefit to moving one of the already placed sensors, it would do that instead of placing a new one. It may turn out though, that it might not be worth placing any of the sensors early, if they moved often enough whenever new terrain is explored. In that method, it might be best to wait till everything explored before placing any sensors. The other option is to use a method which is not a greedy approximation. This survey problem is like the art-gallery problem, which is a form of minimum set cover problem. Many different solutions for this problem have been investigated. They use many methods including hybrid genetic algorithms [16], the Vapnik-Červonenkis dimension of set covers [6], and integer linear programming [3]. There may be

issues with the computational complexity of these methods. It might not be possible to effectively run them in a simulation of a real time robotic system, not to mention on actual modular robot hardware. They will also most likely have similar issues due to the terrain being unknown at first and explored incrementally, and might have to constantly recalculate sensor locations as new terrain is revealed.

6.1.3 Task Progression

The Place Sensor threshold decides when a Blob will progress from the Explore task to the actually placing a sensor. It uses only percentage of unsurveyed area explored to make this decision. This does not take into account whether any of that area is any good to place a sensor. It also bases that on some assumptions that do have flaws. First, the sensor estimate is not perfect, and often too much or too little is estimated to be surveyed (See Figure 3.10). Also, some assumptions are made that may not be valid in some cases. Each blob knows how big the map is, and how much of it is completely unreachable, before the simulation even starts. It uses those numbers to calculate the percent of the non-surveyed map which is explored. In some bounded environments, it may not be a bad assumption, but basing a decision making threshold on it is not the best idea. These numbers are needed to calculate final survey and explore percentages, but allowing the system that controls robots to use them can cause problems. A real world system usually can't intrinsically know those things.

Different methods to transition from exploration to surveying could be used. Instead of deciding based on a specific ratio of terrain explored of the total, it could use another method. Instead of a ratio, it could weight each cell it has explored but not surveyed by a crude estimate of how much it thinks it might cover. This weight might include the height of the cell, its slope and nearby cell values. The values from all cells explored are summed, and when the sum is above a certain threshold, it could choose the best location for a sensor. In that method as well as the current one, calculating when to place a sensor, is much less computationally complex then calculating the location where to place it. If computational resources became less of an issue, it could have a generally lower threshold, and more frequently find the best greedy

sensor location. Instead of immediately deciding it must place a sensor, it can decide if the location is not good enough, and not place the sensor. It will then wait till more of the map is explored before trying again. This increases computation by adding a second decision step to the process, but could prevent some sensors from being placed in poor positions.

6.1.4 Improving Cooperation

Cooperation between blobs is key for Self reconfiguring robots to outperform classical robots. One problem with cooperation occurred when Blobs merge after failing to get to a difficult goal. They don't remember where or what it merged for, so it depends on the internal motivation of the merged blob to find that out. This doesn't always work, because while merging, both blobs are moving toward a center point between them. The local environment is often different enough that somewhere else is chosen to be the next goal, instead of the difficult location it merged for. There are two rather easy ways this could be solved. First, the Blob that needed help could remember the task it was doing before it tried to merge. After the merger is complete, that task could be reinitialized, with a higher than normal motivation. Another option is to have the Blob that asked for help, stay in place instead of moving towards the blob helping it. The helping blob then has to travel the full distance to merge. The tasks of the blob that asked for help are transferred to the new merged blob. This time, the location is the same as before, so the motivation values should be very similar. It should choose either the same goal, or perhaps a better one that has just been discovered. Merging this way approximately doubles the time it takes for a merge to happen, but the energy used to merge stays the same (assuming two equal sized blobs).

In these experiments, the cooperation mechanisms were quite simple and generalized for all circumstances. A blob asks for general Help at its location when it's having difficulty, and another blob responds by having its entire mass merge with it. There are cooperation techniques that could be used that would be an improvement over what is being used currently. More advanced algorithms could be used to foster cooperation in specific scenarios. For example, if a Blob has climbed up onto something it knows is a flat area like a table, it is easy to traverse the top of it, but an issue getting up and

down. Once on top, the blob could immediately split into several small blobs, tasked with exploring the edges and center of the table. A meet up location at the opposite edge it climbed up is planned, and after completing their short exploration, they merge back together again and climb down. The blobs would act much more efficiently during that situation, but only during that situation, as the algorithm is tailored very specifically.

Cooperation could also be made more efficient if blobs more intelligently split and merged. Instead of always splitting in half based on a minimum split size, a blob could make estimates of the difficulty of the map at certain points. When a blob asks for help, it knows the slopes of the cells that gave it trouble. It can use this information to calculate how many extra modules would be needed to get over the obstacle. It could include this in its cry for help, and when another blob responds to it, it can send only the required modules over instead of sending everything.

Bibliography

[1] W. Burgard, M. Moors, D. Fox, R. Simmons, and S. Thrun. Collaborative multi-robot exploration. *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, 1:476–481 vol.1, 2000.

[2] Zack Butler and Eric Fabricant. Reconfigurable Teams: Cooperative Goal Seeking with Self-Reconfigurable Robots . *Distributed Autonomous Robotic Systems*, 8:417–428, 2009.

[3] K. Chakrabarty, S.S. Iyengar, Hairong Qi, and Eungchun Cho. Grid coverage for surveillance and target location in distributed sensor networks. *Computers, IEEE Transactions on*, 51(12):1448–1453, Dec 2002.

[4] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of a*. *J. ACM*, 32(3):505–536, 1985.

[5] Santpal Singh Dhillon and Krishnendu Chakrabarty. Sensor placement for effective coverage and surveillance in distributed sensor networks. *Proc. of IEEE Wireless Communications and Networking Conference*, pages 1609–1614, 2003.

[6] J. González-Banos, H. Latombe. A randomized art-gallery algorithm for sensor placement. *SCG '01: Proceedings of the seventeenth annual symposium on Computational geometry*, pages 232–240, 2001.

[7] Herman Haverkort, Laura Toma, and Yi Zhuang. Computing visibility on terrains in external memory. *J. Exp. Algorithmics*, 13:1.5–1.23, 2009.

[8] Andrew Howard, Maja J Mataric, and Gaurav S Sukhatme. Mobile sensor network deployment using potential fields: A distributed, scalable solution to the area coverage problem. *Proceedings of the 6th International Symposium on Distributed Autonomous Robotics Systems (DARS02).*, pages 299–308, June 2002.

[9] Marc Van Kreveld. Variations on sweep algorithms: efficient computation of extended viewsheds and class intervals. *In Proc. 7th Int. Symp. on Spatial Data Handling*, pages 13–15, 1996.

[10] Haruhisa Kurokawa, Kohji Tomita, Akiya Kamimura, Shigeru Kokaji, Takashi Hasuo, and Satoshi Murata. Self-reconfigurable modular robot m-tran: distributed control and communication. *RoboComm '07: Proceedings of the 1st international conference on Robot communication and coordination*, pages 1–7, 2007.

[11] In So Kweon and Takeo Kanade. High-resolution terrain map from multiple sensor data. *IEEE Trans. Pattern Anal. Mach. Intell.*, 14(2):278–292, 1992.

[12] Lynne E. Parker. Alliance: An architecture for fault tolerant multi-robot cooperation. *IEEE Transactions on Robotics and Automation*, 14:220–240, 1998.

[13] Ioannis M. Rekleitis, Gregory Dudek, and Evangelos E. Milios. Multi-robot exploration of an unknown environment, efficiently reducing the odometry error. *In Proc. of the International Joint Conference on Artificial Intelligence (IJCAI*, pages 1340–1345, 1997.

[14] Langer Rosenblatt, , D. Langer, J. K. Rosenblatt, and M. Hebert. A reactive system for off-road navigation. *In Proceedings of IEEE Conference on Robotics and Automation*, 1994.

[15] Daniela Rus, Zack Butler, Keith Kotay, and Marsette Vona. Self-reconfiguring robots. *Commun. ACM*, 45(3):39–45, 2002.

[16] Jae-Hyun SEO, Yong-Hyuk KIM, Hwang-Bin RYOU, Si-Ho CHA, and Minho JO. Optimal Sensor Deployment for Wireless Surveillance Sensor Networks by a Hybrid Steady-State Genetic Algorithm. *IEICE Trans Commun*, E91-B(11):3534–3543, 2008.

[17] Sascha A. Stoeter, Paul E. Rybski, Michael D. Erickson, Maria Gini, Dean F. Hougen, Donald G. Krantz, Nikolaos Papanikolopoulos, and Michael Wyman. A robot team for exploration and surveillance: Design

and architecture. *In Proc. of the Intl Conf. on Intelligent Autonomous Systems*, pages 767–774, 2000.

[18] J.W. Suh, S.B. Homans, and M. Yim. Telecubes: mechanical design of a module for self-reconfigurable robotics. *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on*, 4:4095–4101 vol.4, 2002.

[19] J. Wood. Landserf, version 2.3. `http://www.landserf.org`.

[20] Brian Yamauchi. Frontier-based exploration using multiple robots. *AGENTS '98: Proceedings of the second international conference on Autonomous agents*, pages 47–53, 1998.

[21] Mark Yim, Kimon Roufas, David Duff, Ying Zhang, Craig Eldershaw, and Sam Homans. Modular reconfigurable robots in space applications. *Auton. Robots*, 14(2-3):225–237, 2003.

Vita

Jacob Hays is a local of Rochester, born on April 25th, 1986. He attended Rochester Institute of Technology in their Undergraduate Computer Science program. He took advantage of RIT's BS/MS program to complement it with a Masters in computer science. His research interests through school were a bit wide, including Computer Graphics, Security, and Artificial Intelligence. For the Masters degree, the focus was in Intelligent Systems, and in particular Self Reconfiguring Robotics. He now works for Cobham Analytic Solutions in Centreville, VA.

Permanent address: 21 Widger Rd.
Spencerport, NY, 14559

This thesis was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.