Carleton UNIVERSITY

FACULTY OF SCIENCE - SCHOOL OF COMPUTER SCIENCE

PRINCIPLES OF COMPUTER NETWORKS
COMP 3203

DISCOVERY OF ROTATING DIRECTIONAL SENSORS

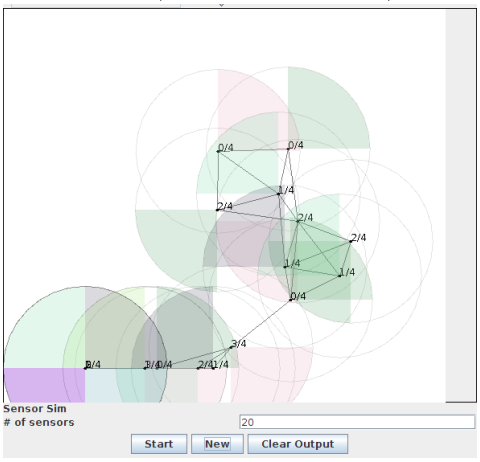
AUTHORS

Wesley Lawrence Danil Kirillov Darryl Hill

PROFESSOR

Dr. Evangelos Kranakis

MONDAY, DECEMBER 3, 2012



Contents

List of Figures

1 Introduction

In this document we examine the results of programmatically simulating algorithms for symmetry breaking for rotating directional sensors. We use the (D,D) model, whereby the sensors have identical transmission and reception beam width. Three algorithms are implemented.

1.1 Antennae Rotation Algorithm

Technically this is the ARAR algorithm denoted in the Neighbor Discovery in a Sensor Network with Directional Antennae in the paper. In this algorithm each sensor rotates one sector, then delays for d_u steps while transmitting and listening for neighbours. In order to properly ensure that the sensors don't a)rotate with the same delay and b)rotate with delays that are multiples of one another, d_u is a prime number based on a colouring of the graph, where $d_u >$ k. This is the only purely deterministic algorithm, so it is guaranteed to finish (ie we have a determinable finish time in addition to expected time). Although it performs tolerably well with wide sensor beams (ie low k values), it is in general not the best performing algorithm. And a particularly dense graph could conceivably run into problems. Most notably that colouring algorithms are NP-hard (though I am sure there are tolerable algorithms for it), but also since the value of the prime numbers needed raises with the density of the graph, two adjacent sensors with high value primes can take a long time to detect one another.

1.2 Random Selection Rotation Mechanism Algorithm

In the RSRMA algorithm chooses between two algorithms, Mech0 and Mech1. Both take two arguments. Mech0 rotates with no sector delay, while Mech1 rotates using a sector delay. RSRMA calls these algorithms with the number of sectors as both arguments. So Mech0 rotates through its k sectors k times with no delay, while Mech1 rotates one sector then delays for k time while sending and listening for signals. At the end of each iteration it chooses Mech0 or Mech1 at random.

Perhaps surprisingly, this is the best performing algorithm of the three detailed here.

1.3 Random Selection Rotation Mechanism Algorithm Prime

RSRMA' operates much the same as RSRMA, except that instead of using k for the second argument it passes in a prime number d. So it will rotate through the sectors d times in Mech0, or rotate with delay d in each sector in Mech1. At the end of each iteration it again chooses Mech0 or Mech1 at random.

The prime numbers are again based on the colouring of the graph, ensuring there is no symmetry in the rotation of the sensors, while the random element improves the run time over ARA. However, it still does not perform as well as RSRMA.

2 Program Breakdown

2.1 Introduction

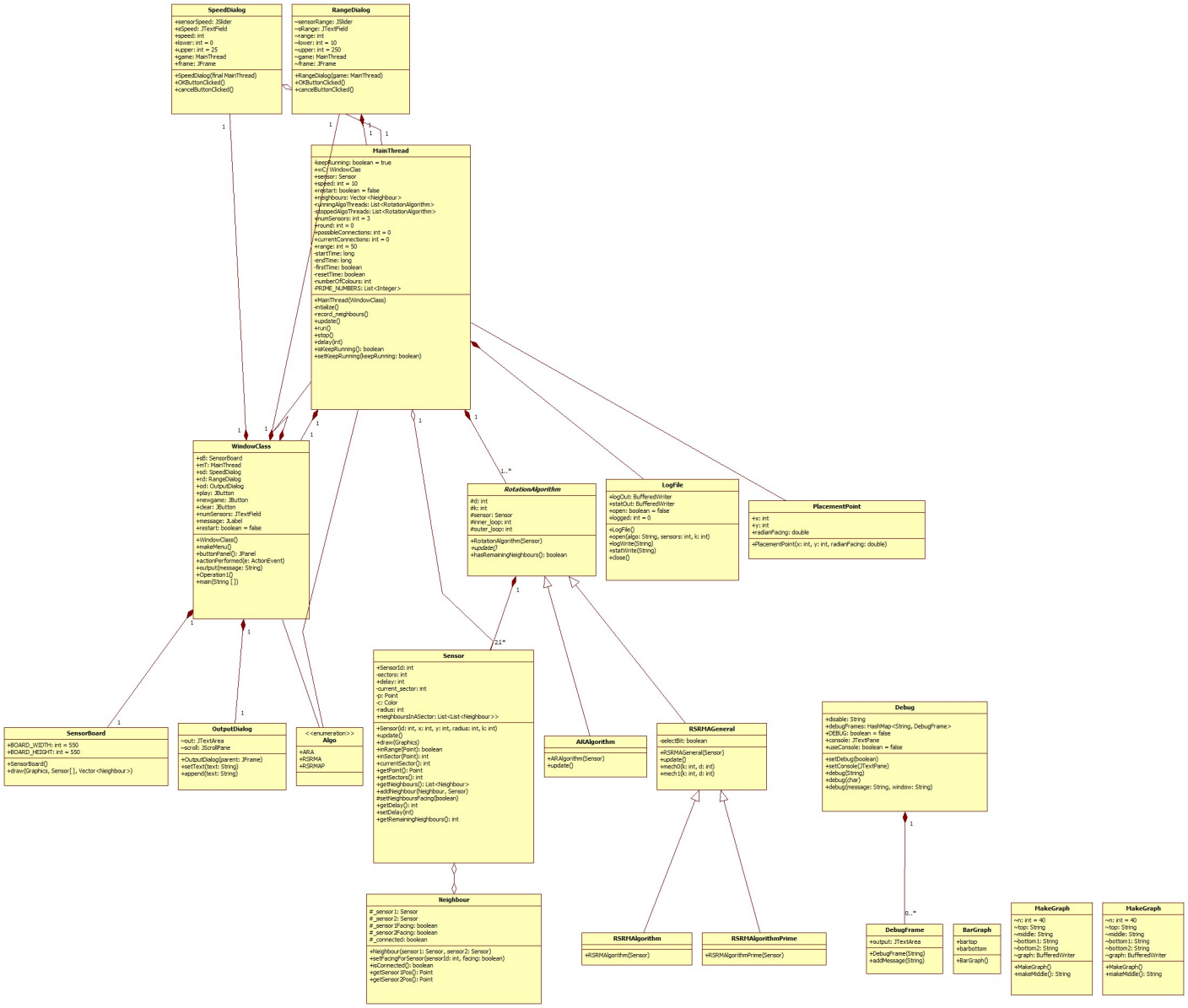
The program is written in java, providing a GUI interface to rapidly enable a wide variety of testing conditions.

2.2 MainThread

The MainThread class is what really makes this application tick. As the name suggest the class provides a thread on which the simulation is ran on. Prior to providing and running a thread, upon initialization, this class does a number of useful things. The first and foremost functionality it carries out is creating all the Sensors. The number of sensors, as well as the number of sectors, and range is determined by parameters from the GUI, modified by the sliders. Creating Sensor can be further broken down into three steps; deciding their placement, identifying the Neighbours, and assign the Sensor to and algorithm. <Something about placement. Either a short description or something saying it's in another section>. Two Sensors are identified as Neighbours if both are in range of each other, falling on some sector of each other. Neighbours are identified in the "record_neighbours" method in the MainThread. This method is ran at most $O(n^2)$ times. Each call to this method passes in the index of a Sensor which needs to have it's Neighbours identified. The method then loops through all the Sensors up until the index passed in, so all the Sensors previously initialized. At each iteration of this loop, it checks if the Sensor at the current iteration of the loop is in range of the Sensor at the passed in index. If they are in range of each other, they are made Neighbours and a prime number is determined from the 168 possible prime numbers for the Sensor at the passed in index, such that no neighbourly Sensors have the same prime number, thus effectively coloring the graph of Sensors both graph theory-wise and literally the Sensor's beam color. After the Neighbours are determined, each Sensor is assigned to an algorithm, this taking O(n) time, as it goes through each Sensor and assigns it to an algorithm. The type of algorithm that all the Sensors will be assigned to is determined by a parameter from the GUI. After the initialization the MainThread is prepared to run. The run method of the MainThread, runs until the application is stopped and does one of the two things and is responsible for spawning the Java thread. It either re-initializes the Sensors if the user has pressed New. The main functionality that the run method provides is calling on update. In update the algorithms are updated, taking an O(n), where n is the amount of algorithms that have not yet been complete. An algorithm is complete when there are no remaining Neighbours for it's Sensor that need to be connected. Once the algorithm is complete it is then moved to another list, that acts an inactive list, so that it is on longer updated. This is to done to significantly improve the overall performance of the application, since the purpose of the simulation is to demonstrate how Sensors discover each other, once a Sensor has discovered all of it's neighbourly Sensors there is no need for it to keep searching/being updated. After the update has happened on the Algorithms, the Neighbours are checked for connection, followed by output to both the GUI and log, and a call to re-draw the sensors on the GUI.

The current approach MainThread takes for "running" the sensors, is that there is a list of algorithms that are each responsible for "running" their Sensors. The algorithms themselves are all updating in the update method of the MainThread, and the MainThread is the only class responsible for spawning one thread that the repeatedly calls on update. The other approach would have been to allow each individual algorithm to spawn their own threads, which would then "run"/update the Sensors, while the MainThread handles checking Neighbour connects and calling on the GUI re-draw. During the development of this application, we have implemented this method. We've found that allowing each algorithm to have it's own separate thread would cause synchronization problems, would model a more practical simulation. However having just one thread in the MainThread and updating the algorithms in the update method that is repeatedly called on, the way that the runs application now, would model a more theoretical simulation, and thus this way was chosen to better demonstrate how the algorithms function and how the Sensors discover each other.

Figure 1: UML Diagram



2.3 GUI Interface

The GUI has some configurable sections. In particular there are slider settings for running speed, the number of sectors and the range. We are technically running the tests in a unit square, so the range has no applicative meaning except in the sense of the visual output (ie it makes bigger circles on screen).

2.4 Known Issues

One issue is with our update loop. Because a sensor is moved and updates its connections in the same loop, it is possible to connect to a sensor that is scheduled to move later in the loop, where technically it shouldn't. In terms of performance it may mean a slightly better performance for every algorithm, but since every algorithm has the same potential advantage the overall comparison is not affected. This will occasionally manifest itself as the final sensors finishing out of alignment in the GUI. It does seem quite rare in practice however.

3 Execution

4 Analysis

4.1 Comparison Methods

Our main means of analysis was to compare algorithm performance over different numbers of sensors and different numbers of sectors. The number of sectors was the same for each sensor for each test, but could vary from test to test.

After building and testing the graphical environment, we set it up to execute batches of tests independant of the graphical interface. This allowed us to collect a lot of data quickly.

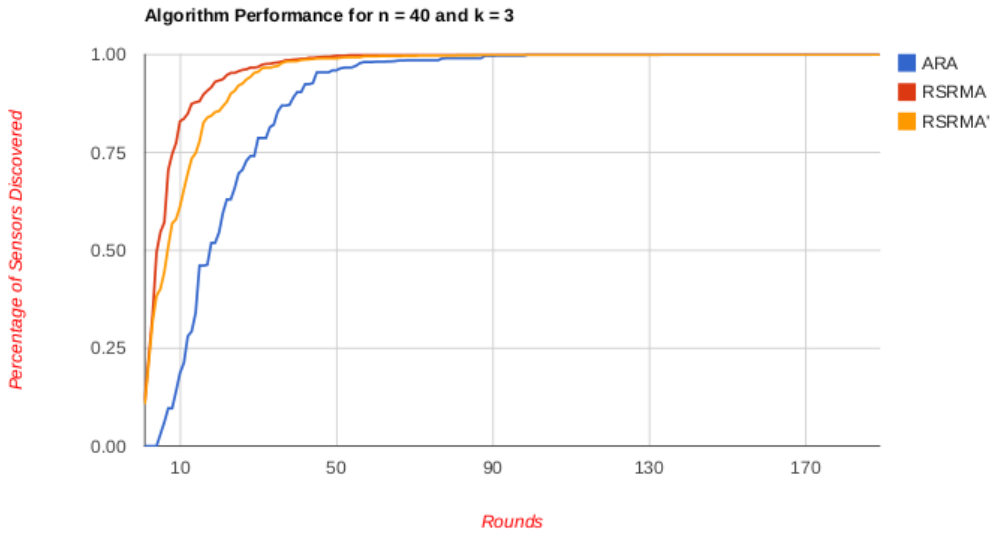
Specifically the batch runs would be for a particular number of sensors passed in as a command line argument. A second argument t is used for the number of tests to be run. The tests are run t times for each algorithm running over the k (number of sectors) values from 3 to 12. Log files are generated as well as statistical files. The statistical files are performance numbers formatted for graph generation. This data is then run through an averaging algorithm which reads in all the files, averages the individual tests while keeping the different sensor and sector values separate, and writes them out to another statistics file. This "average" statistics file is then used to generate a graph (a number of which are included in this write-up).

4.2 Comparision

As you can see in figures 2, 3, and 4, the highest performer at low numbers of sensors is RSRMA. If you remember, it passes in the number of sectors as its second argument and chooses between Mech0 and Mech1. The primary method of symmetry breaking of this algorithm is the random selection of either Mech0 or Mech1. Interestingly, for all the effort put into developing algorithms guaranteed to break symmetry, apparently sometimes it pays to simply roll the dice.

You may also note that as the number of sectors decreases (ie the sector beam gets wider), ARA gets a noticable performance boost in comparison to the other two algorithms.

Figure 2



At extremely low numbers of sensors, such as 10 (5, 6, and 7) RSRMA and RSRMA' are difficult to distinguish. RSRMA starts out a bit quicker but by the end they are neck and neck. ARA is again the

Figure 3

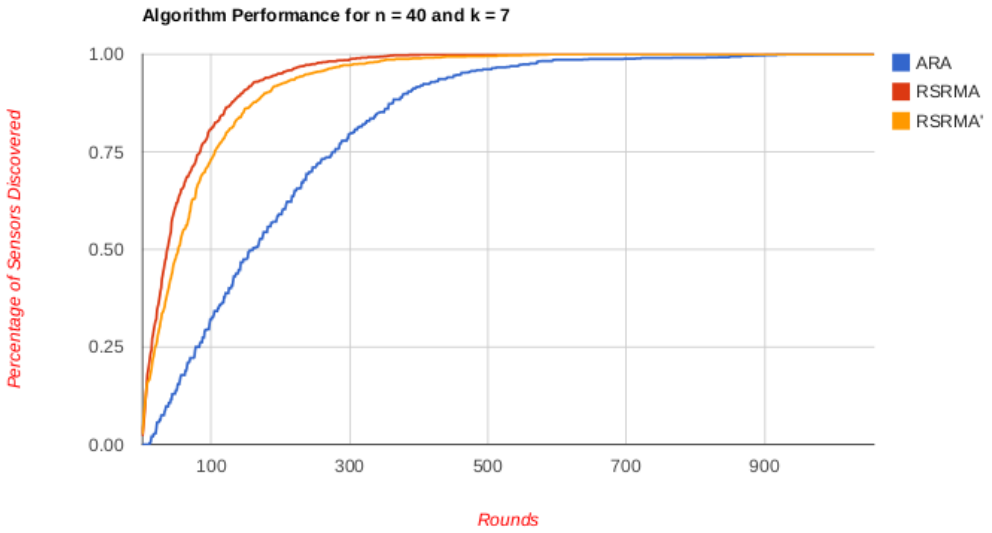
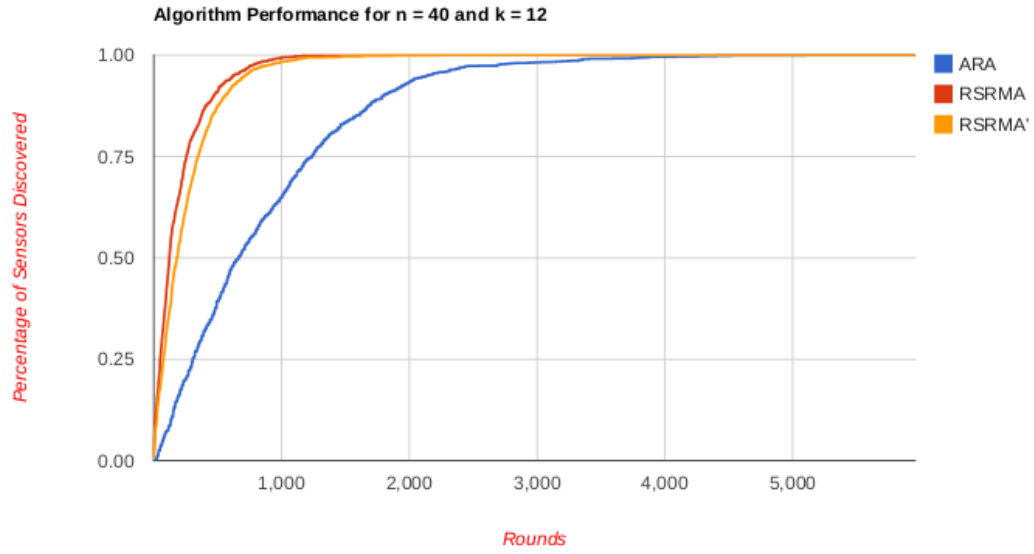


Figure 4



poor performer, but gets a boost at low k values.

Figure 5

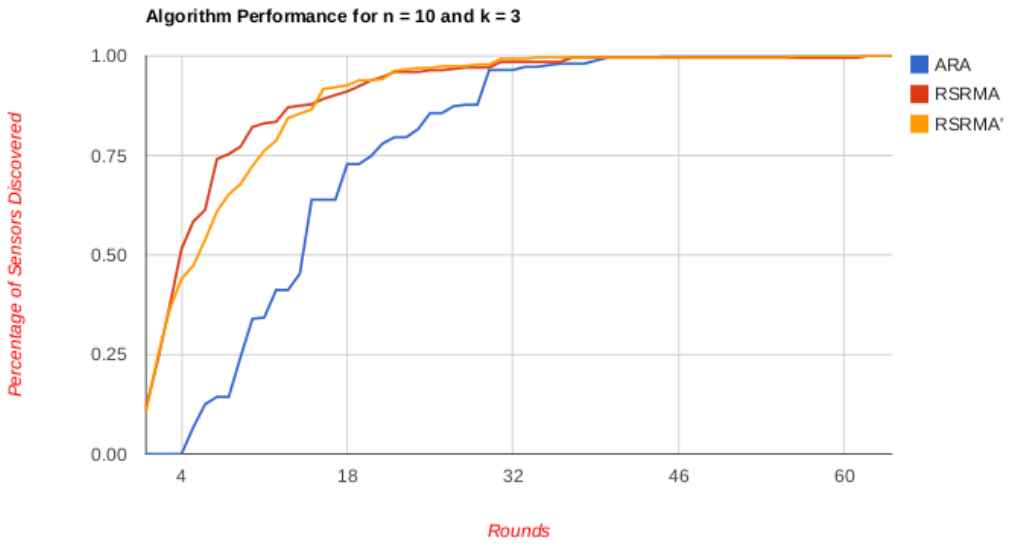


Figure 6

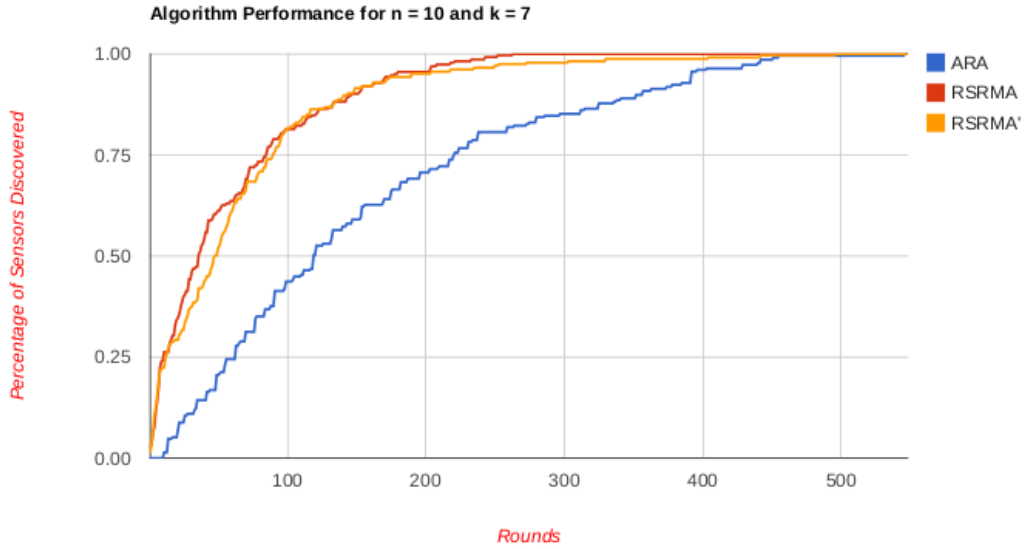
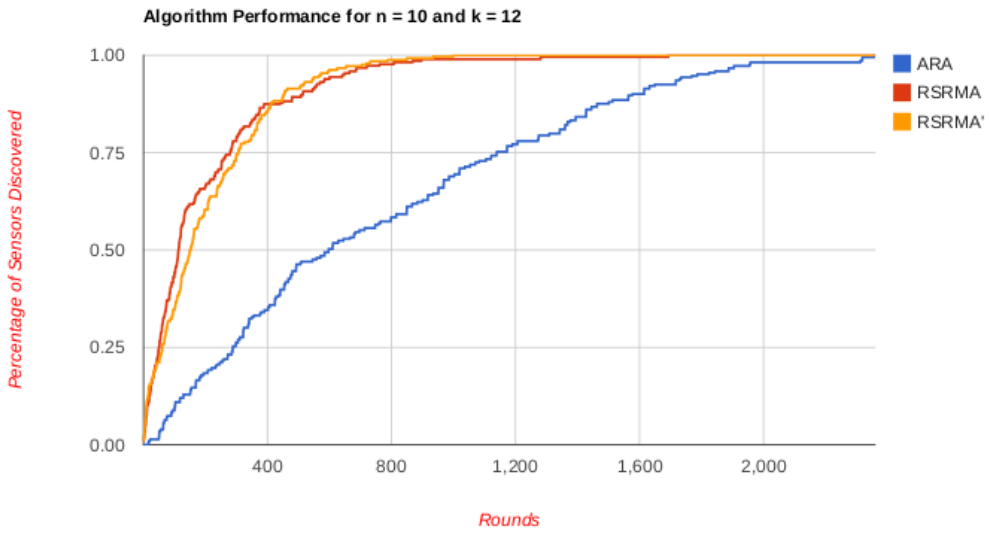


Figure 7


Algorithm Performance for n = 10 and k = 12
Percentage of Sensors Discovered
1.00
0.75
0.50
0.25
0.00
400
800
1,200
1,600
2,000
Rounds
ARA
RSRMA
RSRMA'

5 Conclusions