Carleton UNIVERSITY

FACULTY OF SCIENCE - SCHOOL OF COMPUTER SCIENCE

PRINCIPLES OF COMPUTER NETWORKS
COMP 3203

DISCOVERY OF ROTATING DIRECTIONAL SENSORS

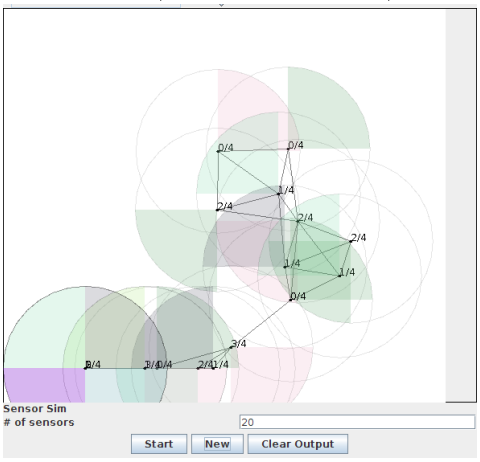
AUTHORS

Wesley Lawrence Danil Kirillov Darryl Hill

PROFESSOR

Dr. Evangelos Kranakis

MONDAY, DECEMBER 3, 2012



Contents

1 Introduction **3**
1.1 Antennae Rotation Algorithm . . . 3
1.2 Random Selection Rotation Mechanism Algorithm . . . 3
1.3 Random Selection Rotation Mechanism Algorithm Prime . . . 3

2 Program Breakdown **4**
2.1 Introduction . . . 4
2.2 MainThread . . . 4
2.2.1 Sensor Placement . . . 4
2.2.2 Neighbour Identification . . . 4
2.2.3 Assigning Algorithms . . . 6
2.2.4 Running the Simulation . . . 7
2.3 Rotation Algorithms . . . 7
2.4 GUI Interface . . . 8
2.5 Known Issues . . . 8

3 Execution **9**
3.1 Normal . . . 9
3.2 No GUI . . . 9
3.3 Testing . . . 9

4 Analysis **10**
4.1 Comparison Methods . . . 10
4.2 Comparision . . . 10

5 Conclusions **20**

List of Figures

1 UML Diagram . . . 5
2 Sensor Placement Example 1 . . . 6
3 Sensor Placement Example 2 . . . 6
4 Rotation Algorithms UML . . . 7
5 Sensors = 10, Sectors = 3 . . . 10
6 Sensors = 10, Sectors = 7 . . . 11
7 Sensors = 10, Sectors = 12 . . . 11
8 Sensors = 40, Sectors = 3 . . . 12
9 Sensors = 40, Sectors = 7 . . . 12
10 Sensors = 40, Sectors = 12 . . . 13
11 Sensors = 100, Sectors = 3 . . . 13
12 Sensors = 100, Sectors = 7 . . . 14
13 Sensors = 100, Sectors = 12 . . . 14
14 Sensors = 1000, Sectors = 3 . . . 15
15 Sensors = 1000, Sectors = 7 . . . 15
16 Sensors = 1000, Sectors = 12 . . . 16
17 Sensors = 5000, Sectors = 3 . . . 16
18 Sensors = 5000, Sectors = 7 . . . 17
19 Sensors = 5000, Sectors = 12 . . . 17

20 Sensors = 100, Sectors = 12, 20 000 rounds 18
21 Sensors = 1000, Sectors = 12, 20 000 rounds 18
22 Sensors = 5000, Sectors = 12, 20 000 rounds 19

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. One is purely deterministic, one is random, and one is a combination of the two.

While a variety of data is analyzed, the main focus of our analysis lies with the effect on graph density on algorithm performance. As the proliferation of wireless networks continues, network density is an everpresent and ever-increasing issue. More density means a greater number of neighbours and a greater number of possible connections per node. We find that the performance of algorithms for sensor discovery that depend on a colouring of a graph begin to suffer as the density of the graph increases.

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. This is most notable in particularly dense graphs. There is the relatively minor issue that colouring algorithms are NP-hard (though I am sure there are tolerable algorithms for it), but mainly the value of the prime numbers used to break symmetry rises with the density of the graph, and two adjacent sensors with high value primes can take a long time to detect one another.

1.2 Random Selection Rotation Mechanism Algorithm

In RSRMA the 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, and seems to suffer little or no negative impact from a rise in graph density.

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, its dependance on prime numbers still provides an achilles heel, and in dense graphs it does not perform as well as RSRMA.

2 Program Breakdown

2.1 Introduction

The program is written in Java for simplicity for and portability. It can be broken down into three main parts. The MainThread is responsible for creating the simulation and updating the algorithms. The RotationAlgorithms (and subclasses) are responsible for updating each sensor when they themselves are updated by the the MainThread. Lastly, the GUI allows users to configure parameters used by the MainThread to create and run the simuations.

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. This class is responsible for multiple things, mainly the initialization of the simulation, and the running of the simulation. First off, 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 sensors can be further broken down into three steps; deciding their placement, identifying the neighbours, and assigning each sensor to an algorithm.

2.2.1 Sensor Placement

Sensor placement is not to a random position, as this can cause two problems. Sensors can be to far away to connect, or so close that they sit almost on top of each other. This was solved by determining valid placements for new sensors upon the placement of a sensor, with the first sensor place in the centre of the simulation space. After placing the first sensor, a random number generator determines (between 3 and 8) how many sensors can be place around the starting sensor. At a random angle, the first valid position is recorded at 90% the sensor range, and then the remaining placements are placed equal-radian apart, surrounding the first sensor. Placement data also contains the position of the sensor that spawned the placement. This is used to help generate new sensors growing outwards. After the initial sensor placement, a valid placement is randomly chosen out of the valid placement list, and a new sensor is placed. For these new sensors, they generate new valid placements, but with a small difference. They generate between 2 and 5 sensors, which are spread out along a 180 arc, which is centred on the new sensor, faces away from the sensor that generated the new sensor. This style of placement generates graphs that are always connected, but that also grow in random directions. It makes it less likely for sensors to overlap, but when generating over 100 sensors, overlaps begin to occur quite frequently.

2.2.2 Neighbour Identification

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 takes at most $O(n^2)$ times, since it runs on n sensors, checking at most, n other sensors. 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 (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 colouring the graph of sensors both graph theory-wise and literally the sensor's beam colour.

Figure 1: UML Diagram

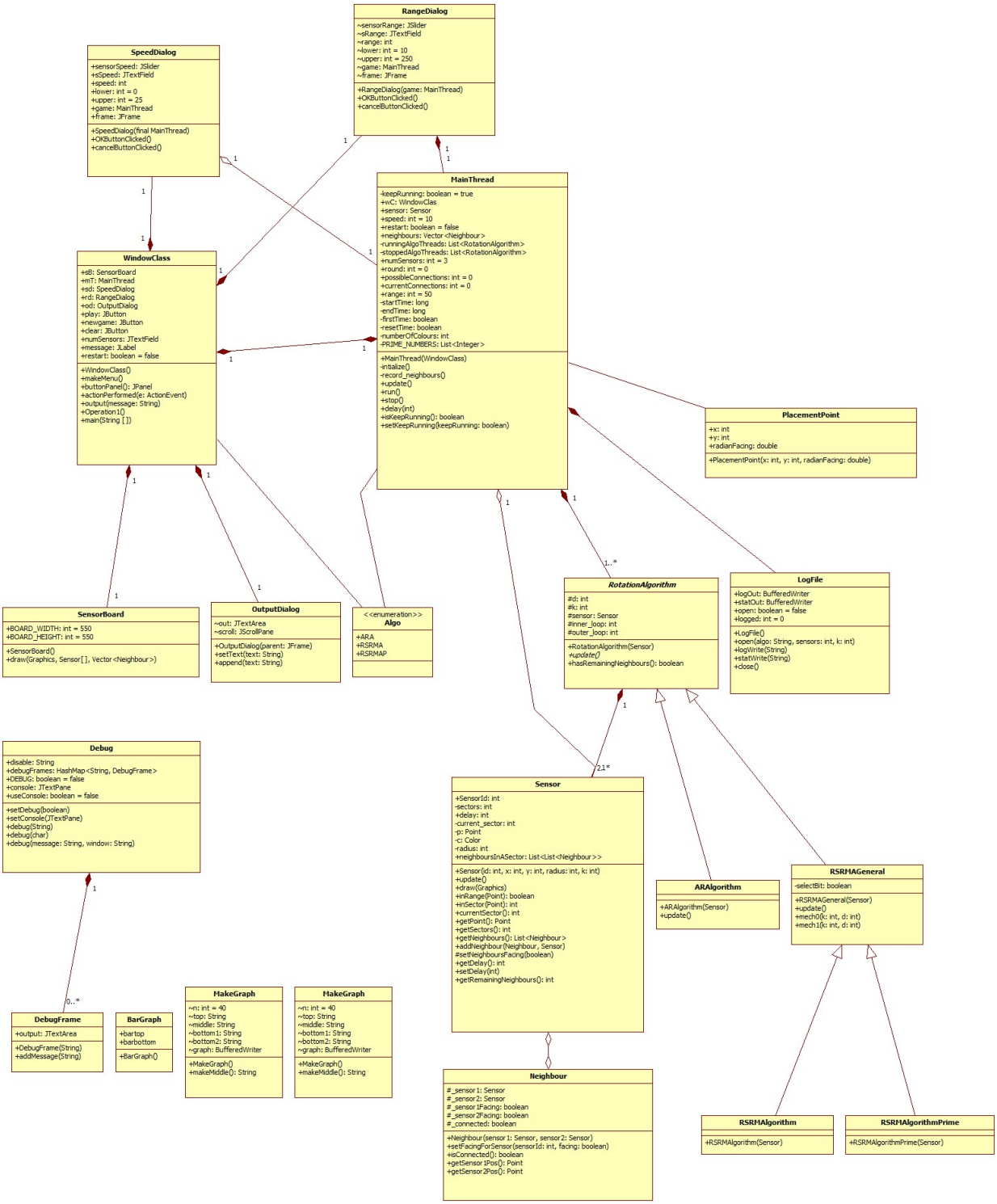
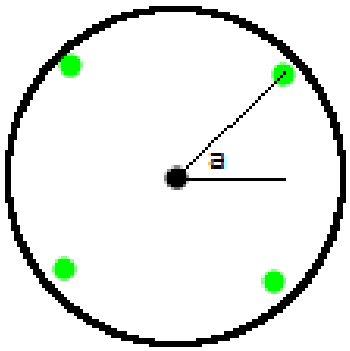
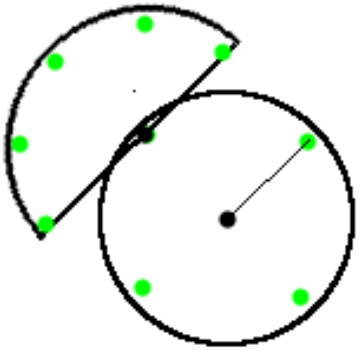


Figure 2: Sensor Placement Example 1



The first sensor has been placed in the centre with placements shown in green. Between 3 and 8 sensors can be chosen to be placed (choose 4), first first at the random angle 'a'. The rest are placed equal-radian apart.

Figure 3: Sensor Placement Example 2



One of the placements was chosen to place a sensor, and is now black. A 180 degree semi-circle, facing away from the sensor that spawned the sensor we just placed (in this case, the original sensor). Between 2 and 5 sensor may be chosen as placements, this time 5. They are place equally distributed along the half circle, as seen with the new green circles. The next sensor can be any green circle, until all sensor are generated.

2.2.3 Assigning Algorithms

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, either by completion, or the user clicking 'Stop'. It either re-initializes the sensors if the user has pressed 'New', or continues to run or re-run the simulation if the user clicks 'Start'. 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. When a sensor is updated, it tells all the neighbour it is facing that it is no longer facing them, it increments to the next sector, and tells it's new neighbours that it is facing them. A neighbour is connected whenever both sensors it is linked to are 'facing'. Sensors stop checking neighbours that have been connected, speeding up their update time. 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 removed from the list. This is to done to significantly improve

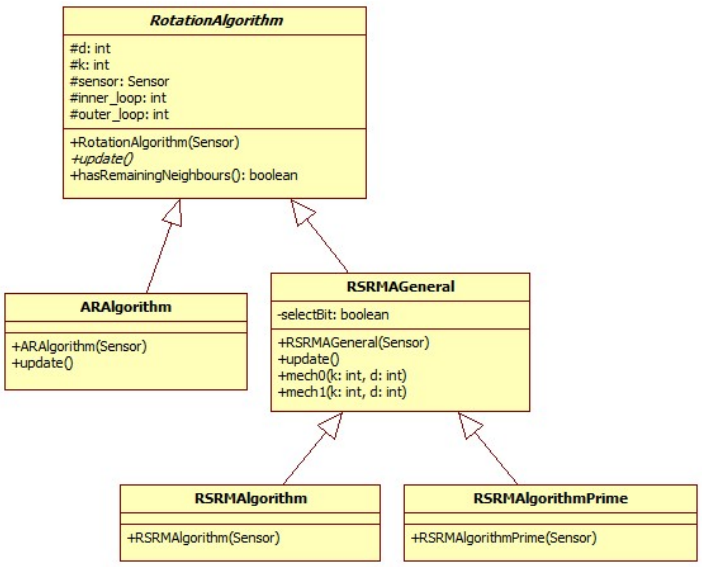
the overall performance of the application, since the purpose of the simulation is to demonstrate how sensors discover each other, once a sensor has discovered all of it's neighbourly sensors there is no need for it to keep searching/being updated. After the update has happened on the algorithms, the 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.

2.2.4 Running the Simulation

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 then 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 had implemented this method. We've found that allowing each algorithm to have it's own separate thread could cause synchronization problems, but would model a more practical simulation. However, having just one thread for MainThread and repeatedly updating the algorithms in the update method, the way that the runs application now, models a more theoretical simulation where everything is updated synchronously, and thus this way was chosen to better demonstrate how the algorithms function and how the sensors discover each other.

2.3 Rotation Algorithms

Figure 4: Rotation Algorithms UML



To implement the rotation algorithms we have decided to use a strategy pattern, as shown in Figure 4, to allow the MainThread to update a RotationAlgorithm without specifically knowing which type it is. The main thread includes a List of RotationAlgorithms that get initialized, based on GUI parameters, to a more specific algorithm such as ARA, RSRMA, or RSRMA' . It was easy to see that a strategy pattern was needed, simply because all the algorithms shared a number of things in common. Using the strategy pattern allowed us to abstract common methods and variables, such as the delay (d), sectors (k)

variables and mech0 and mech1 methods. This produced more elegant code. Each algorithm takes in a Sensor that it will be responsible for rotating, and based on that sensor, it determines the algorithm's delay (d) and number of sectors (k). Because these algorithms do not spawn their own threads and rely on the MainThread to loop through them, the delay loops were done slightly differently. The update method for each algorithm is responsible for rotating the Sensor. In the update methods the outer_loop and inner_loop variables are used to simulate delay. In the case of the ARAlgorithm, where the original algorithm is described in the assignment and the additional papers, the for-loop inside of the while-loop acted as a delay. It was implemented so that the while-loop was the MainThread update loop and the inner for-loop was represented by the outer_loop variable, to cause delay. To be more specific, the outer_loop variable upon initialization of an ARAlgorithm is set to the sensor's 'd', it's color/prime. Every time the update happens for the ARAlgorithm, the outer_loop variable decrements, until it reaches zero. When it is zero the sensor updates, causing a turn, and the outer_loop variable is re- initiated to the sensor's 'd'. This simulates the delay in the algorithm, and although the loop does not look the same, it functions the same way. For the RSRMA and RSRMA' the loops were handled in a similar fashion. However each time the outer_loop variable reached zero, it would randomly pick between mech0 and mech1. With the level of abstraction that was made by implementing the strategy pattern, the RSRMAGeneral contained the majority of how the randomized algorithms functioned. It was noticed that the difference between RSRMA and RSRMA' was that one passed in two 'k', treating one of the 'k's as a 'd', while the other passed in 'k' and 'd' to the mechs. Thus the only code that RSRMAlgorithm and RSRMAlgorithmPrime possess is their constructors, where the former assigns the Sensor's 'k' value to its 'k' and 'd', and the later assigns its Sensor's 'k' and 'd' values to it's own 'k' and 'd' values, respectively.

2.4 GUI Interface

The GUI provides the ability to configure the tests being run. In particular there are slider settings for delay in milliseconds, 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), but it will also have an impact on graph density (ie higher range values, as a fraction of the unit square, will mean more connections and a denser graph). The selection of the colouring of the sensor beams is based on the "colouring" of the graph, so that sensors that have the same colouring (ie prime number delay) will be dislayed with the same colour onscreen.

In addition there is a plain output window that shows the same text that is being output to the logfile, including the number of connections made and the number of connections left to be made. This allows you to monitor the state of the algorithm as it is running.

2.5 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

3.1 Normal

The normal state of the program is to run with the gui interface. This allows you to run tests in a singular fashion, monitoring both the written and visual output for performance and any other notable statistics. While it is being run there are logfiles and statfiles being generated. The logfiles are for human consumption, that is the events are written out in explicit english, while the statfiles are numbers formatted specifically for generating graphs.

3.2 No GUI

If the program is run with command line arguments, $-t < \#oftests > -s < \#ofsensors >$, you can run batches of tests without the gui at all. It will produce a batch of sensors which it will then run for all three algorithms and every k value (number of sectors) from 3 to 12. This allows for a fair comparison of algorithm performance. For each test number of new batch of sensors is generated, and the entire process is repeated, generating log and statfiles. So, for example, if you ran a batch of 10 tests, each test would run the full number of k values (10) for each algorithm ($\times 3$), and repeated across 10 tests ($\times 10$). So in all 300 tests would be run generating the appropriate log and stat files (which are later amalgamated into graphs).

3.3 Testing

This is admittedly one of our weaker areas, in the sense of having well planned test cases that are their own programs. There are JUnit tests for the graph colouring (it generates a batch of sensors and ensures that no two neighbours share a colour. While not a rigorous proof of correctness, run repeatedly for dense graphs it is statistically highly unlikely to produce a false result) and also a JUnit test to ensure proper matching of sensor sectors.

There is also a Debug class, whose main purpose is to provide a class that will print out debug statements, but with the ability to disable it from within the class itself, rather than at each individual print statement (very similar to qDebug(), if you are familiar with it). So a lot of testing revolved around the less formal use of the print statements (which are not as easy to document).

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 3, 7 and 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

It should be noted that the jagged graphs are a unfortunate side effect of running low numbers of sensors. Since the tests finish so quickly there are a lot of discrete values. Even trying to average over as many as 1000 tests did nothing to alleviate it.

As you can see in figures 8, 9, and 10, 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 (though it never actually catches up).

Figure 5: Sensors = 10, Sectors = 3

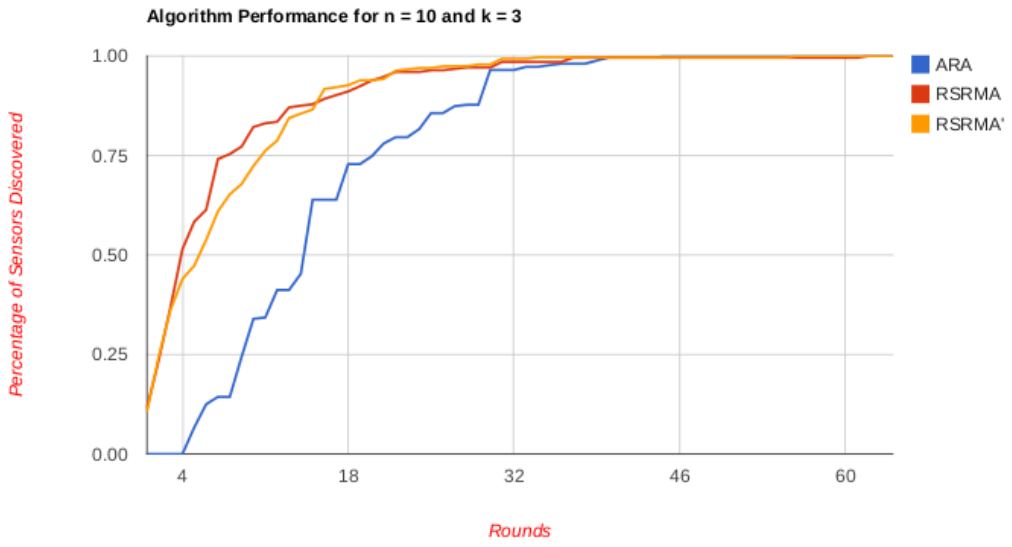


Figure 6: Sensors = 10, Sectors = 7

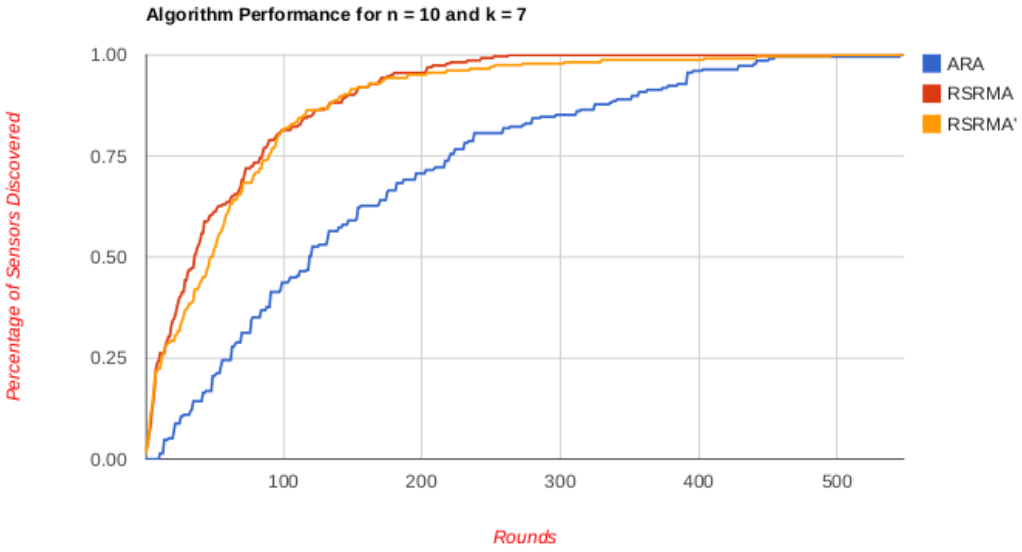
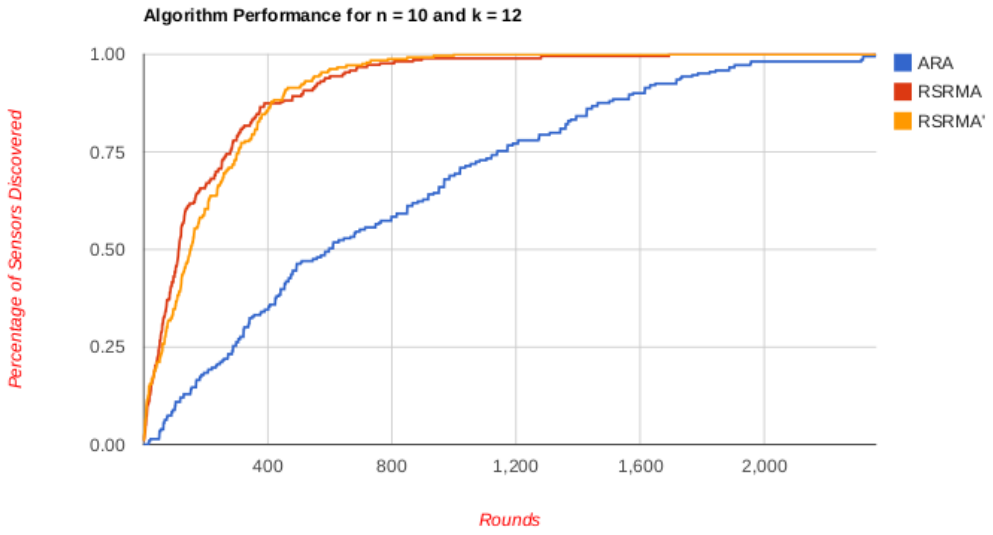


Figure 7: Sensors = 10, Sectors = 12



At extremely low numbers of sensors, such as 10 (figures 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 poor performer, but gets a boost at low k values.

At 5000 sensors it takes upwards of 90000 rounds to finish. But at 90000 rounds the graphs are not as informative, so we limited rounds to 20000.

Every test did finish, but higher sensor values produce denser graphs, and denser graphs produce higher prime numbers, which means a longer overall time to finish. So the higher the sensor count, the more dense the graph, and the performance of the algorithms that are dependant on prime numbers really

Figure 8: Sensors = 40, Sectors = 3

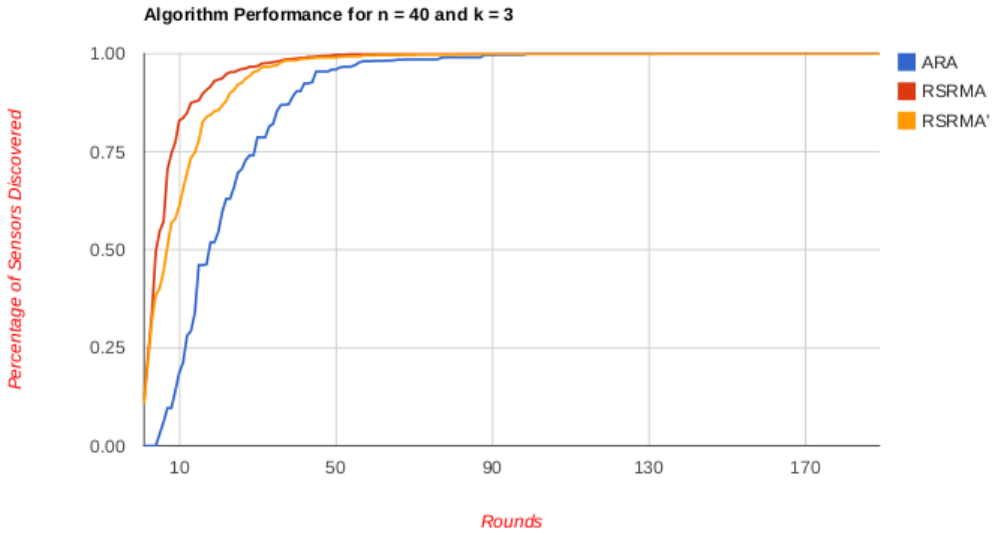
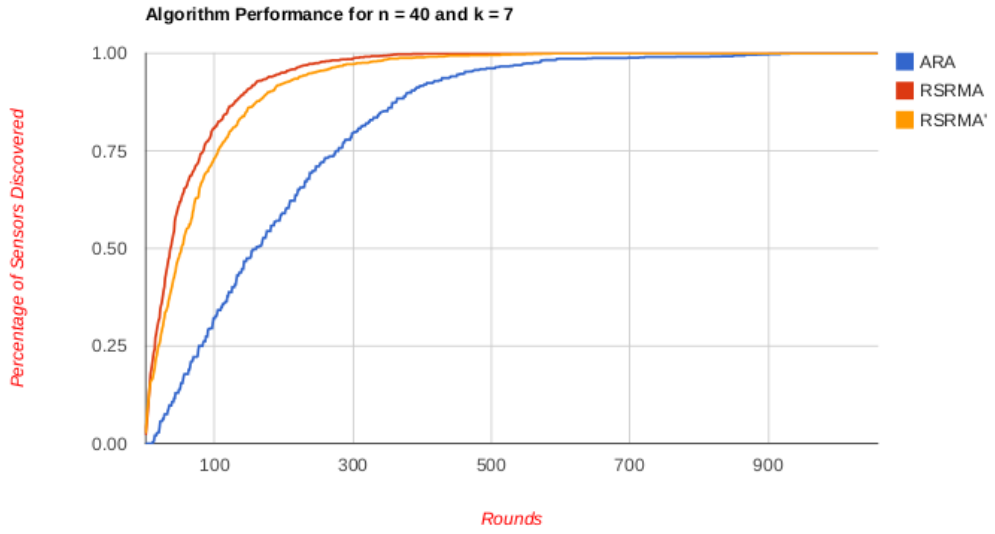


Figure 9: Sensors = 40, Sectors = 7



suffers. RSRMA finishes 5000 sensors in roughly the same time as 1000 sensors, so clearly randomized algorithms hold an advantage in dense graphs.

Figure 10: Sensors = 40, Sectors = 12

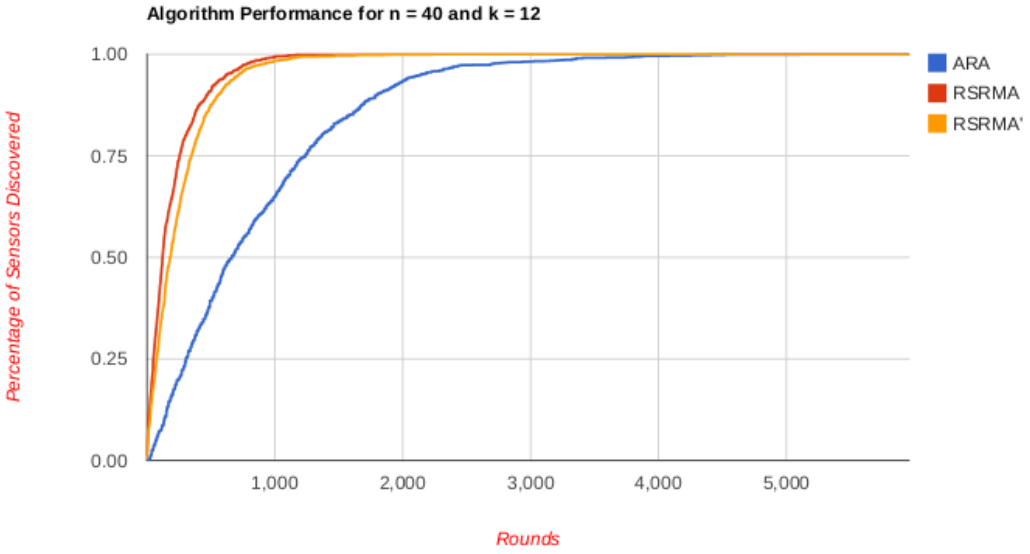
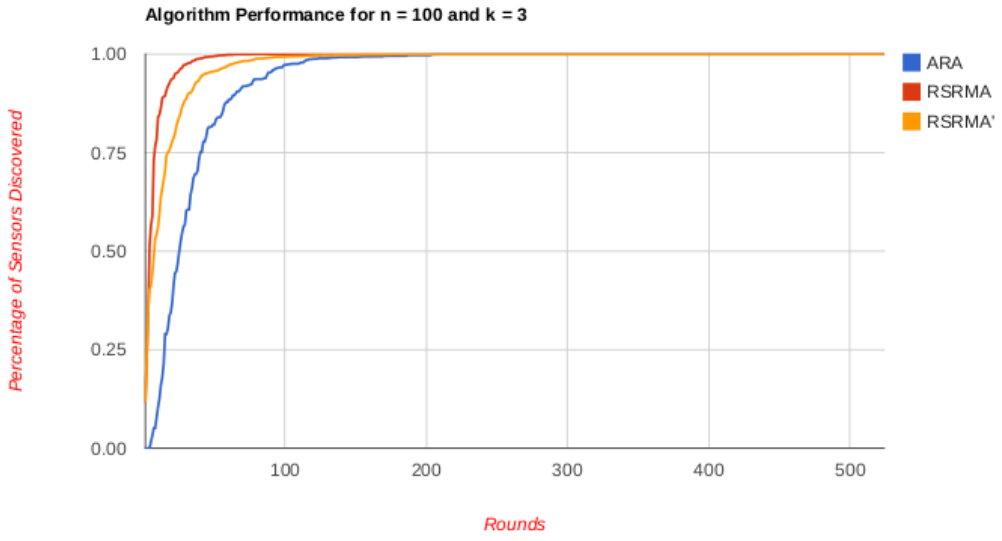


Figure 11: Sensors = 100, Sectors = 3



Some of the graphs are deceptive, since they show the performance over varying amounts of time. For graphs that compare varying k values this was deemed preferable, since if we fix the time at whatever k=12 finished at, the graphs for k=3 essentially appear to shoot straight up. So while in some ways it is a sloppy comparison, it seemed the lesser of two evils.

We did three additional graphs of k = 12 of varying density in figures 20, 21, and 22, where we fix the number of rounds to 20 000. We can see the performance difference in the three algorithms as the density increases. In particular, the performance of RSRMA does not suffer, while the performance of the algorithms dependant on a graph colouring and prime numbers suffers considerably in comparison.

Figure 12: Sensors = 100, Sectors = 7

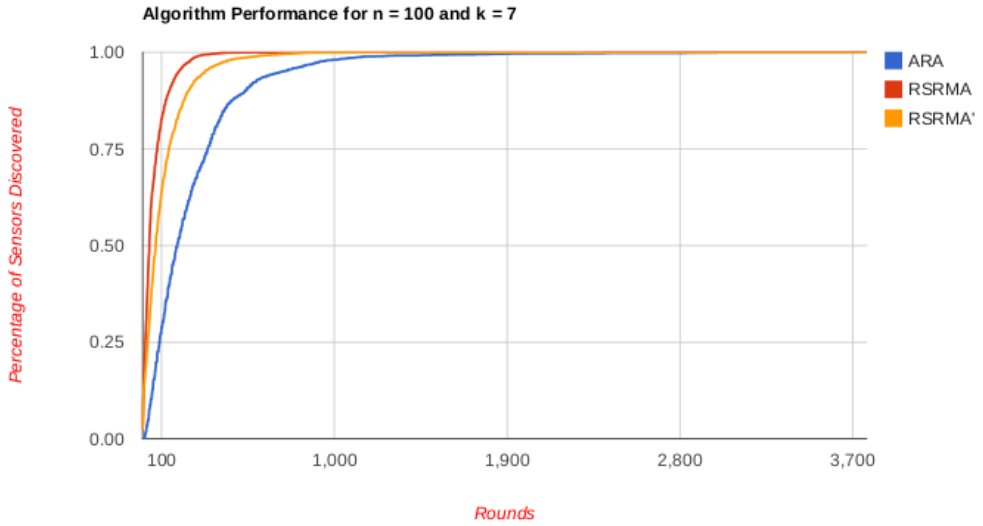


Figure 13: Sensors = 100, Sectors = 12

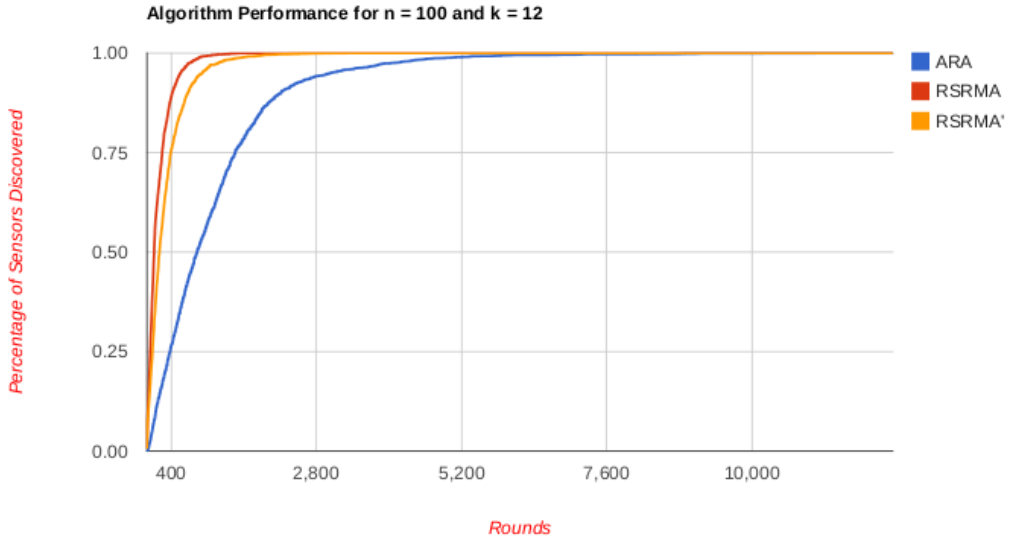


Figure 14: Sensors = 1000, Sectors = 3

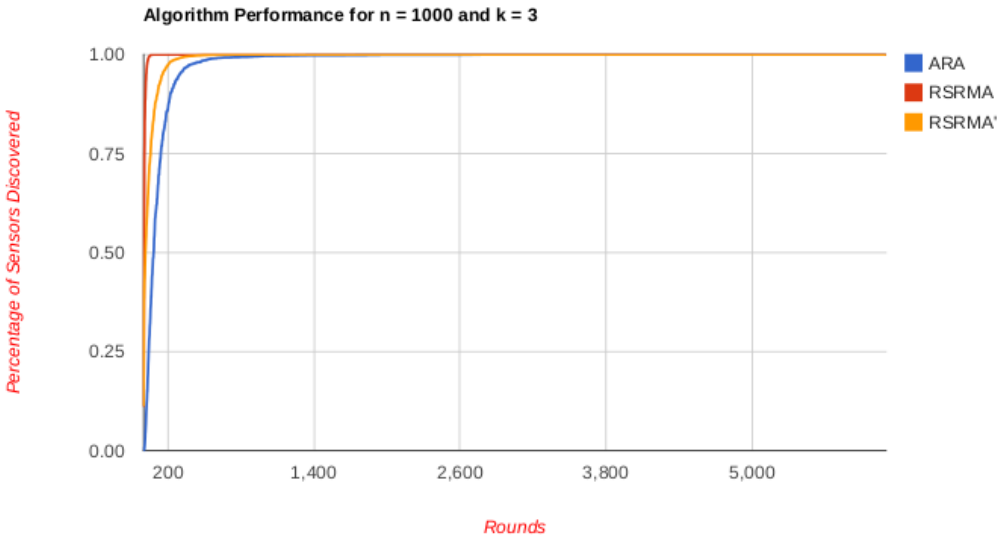


Figure 15: Sensors = 1000, Sectors = 7

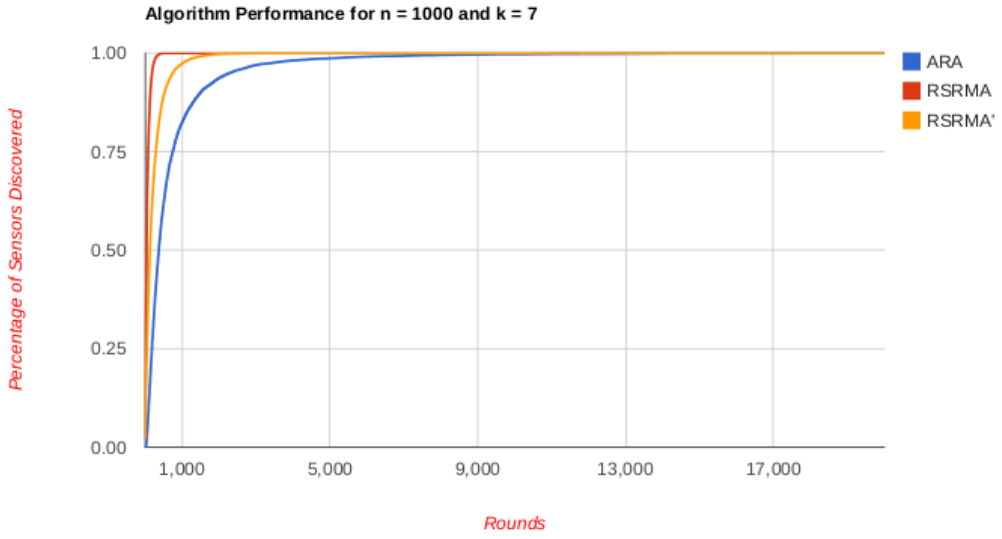


Figure 16: Sensors = 1000, Sectors = 12

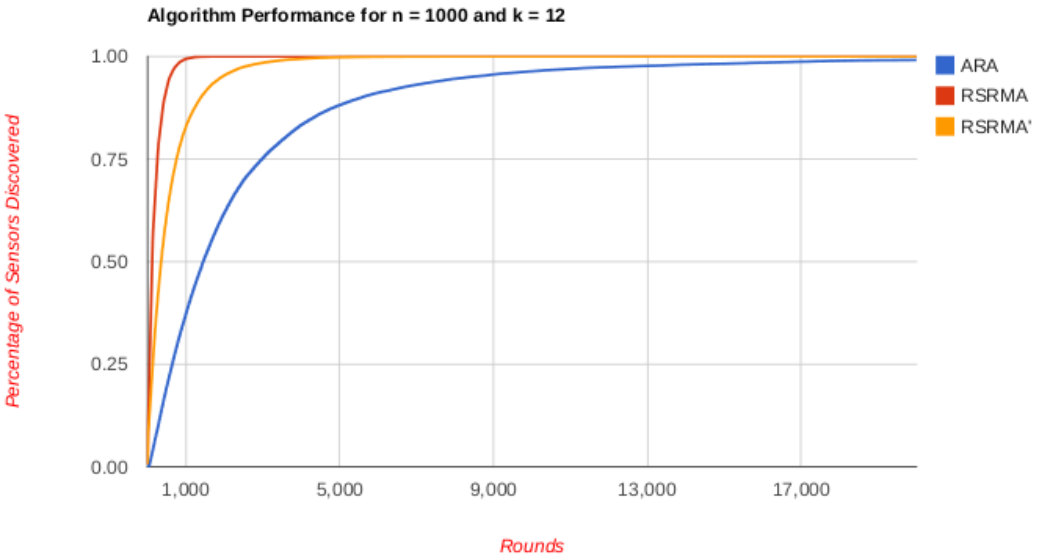


Figure 17: Sensors = 5000, Sectors = 3

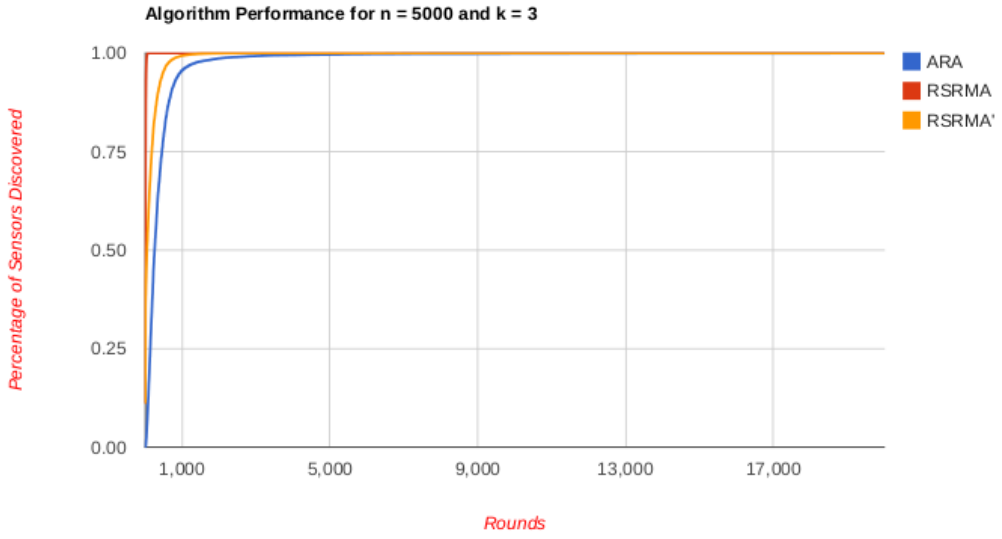


Figure 18: Sensors = 5000, Sectors = 7

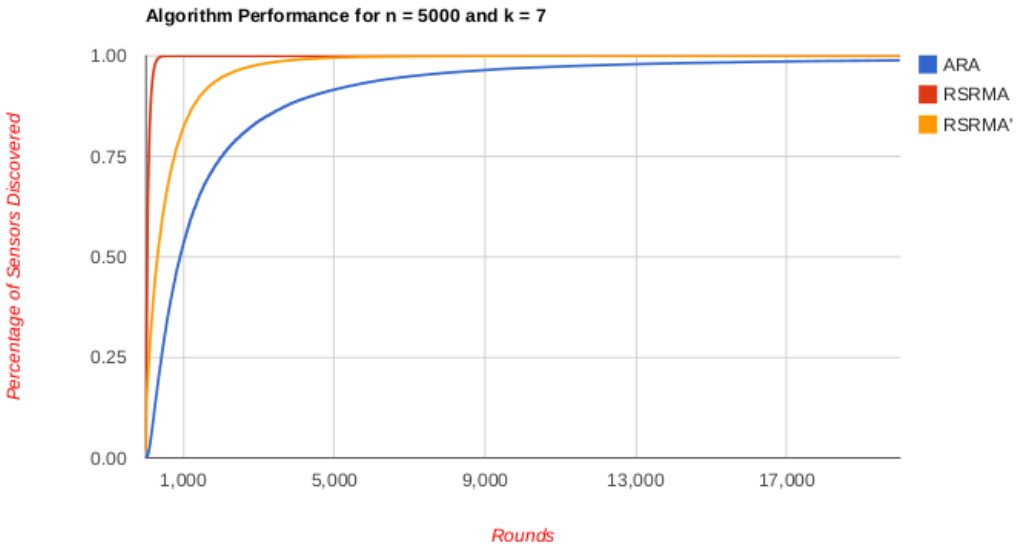


Figure 19: Sensors = 5000, Sectors = 12

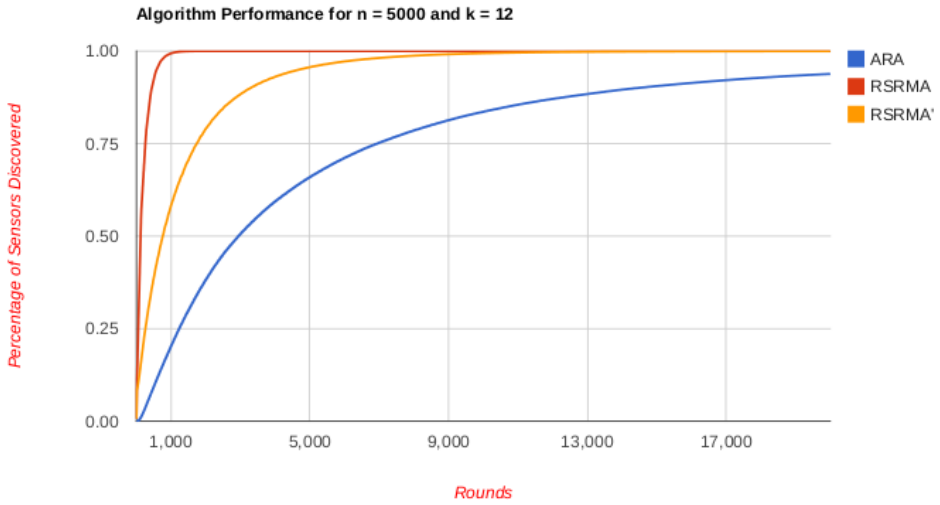


Figure 20: Sensors = 100, Sectors = 12, 20 000 rounds

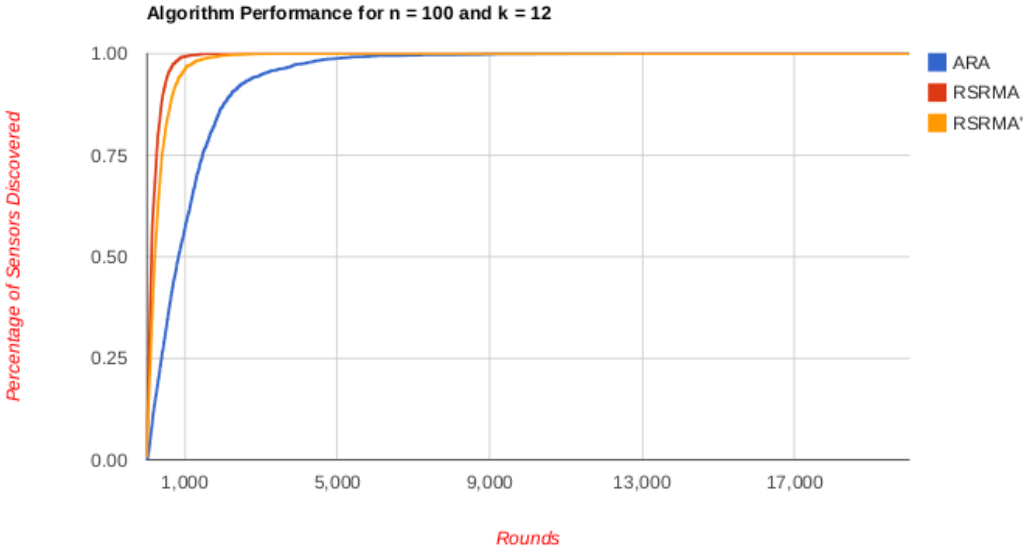


Figure 21: Sensors = 1000, Sectors = 12, 20 000 rounds

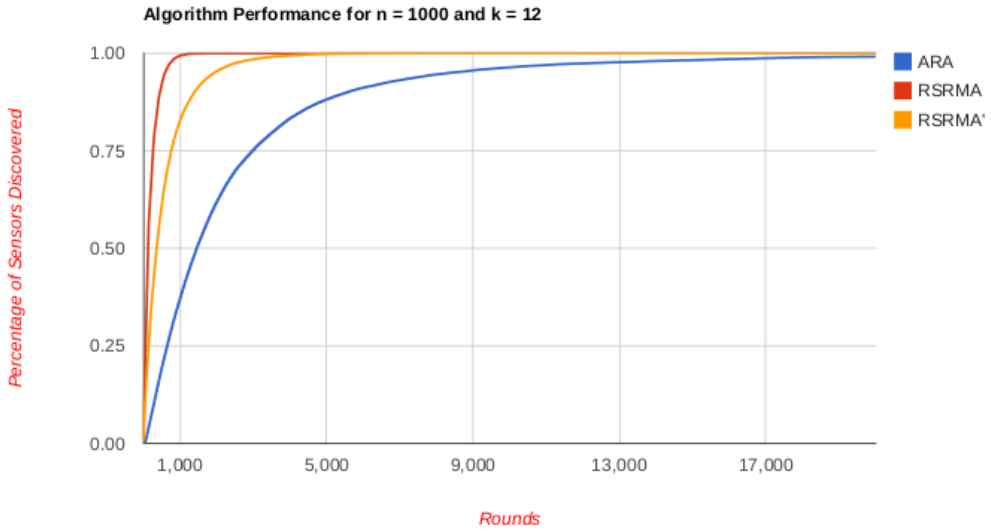
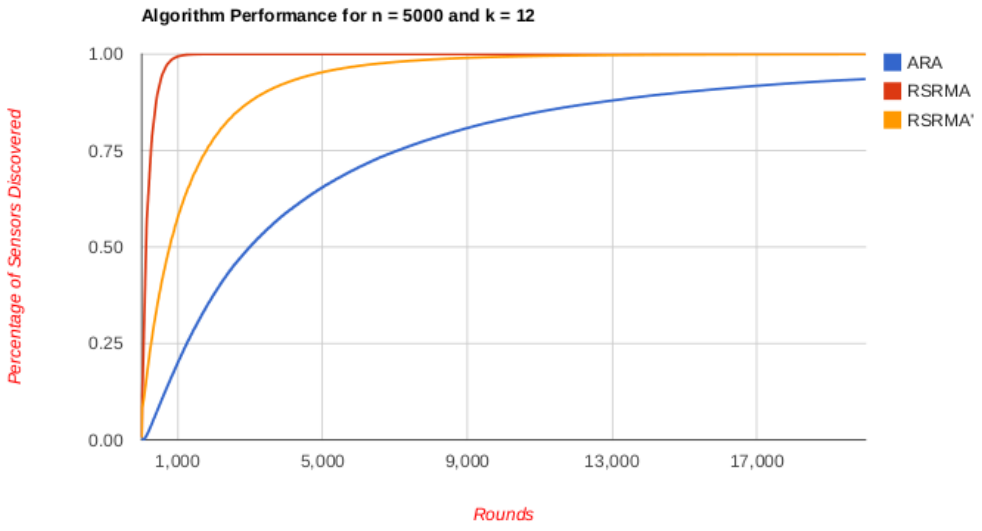


Figure 22: Sensors = 5000, Sectors = 12, 20 000 rounds



5 Conclusions

In overall algorithm performance, RSRMA is the clear winner. Not relying in global knowledge means it is versatile; it can be implemented anywhere, no just where there is global network knoweledge, and the overall simplicity of the algorithm provides another significant advantage. Moreover, its performance remains consistent even in dense graph situations, and/or with high k values. It is a solid performer in all areas and, relative to the other two algorithms implemented here, has no drawbacks.

ARA is purely deterministic. With a low number of sectors and a sparse graph it is a servicable algorithm. However, relying purely on prime numbers to break symmetry limits its usefulness in comparison to the other algorithms. In the situation of a high sector count (ie narrow sensor beam width) and/or a dense graph, we are forced to use higher and higher prime numbers to break symmetry, and the performance suffers.

RSRMA' tries to combine the two approaches. It too suffers in dense graph situations or with high k values, due to high prime number cycles. In a combination of sparse graphs and low k values, where the prime numbers remain low, it is comparable to RSRMA. However, as the density and k value increase, the performance falls away. The randomized element means it does not suffer as much as ARA, and the determistic element guarantees completion. However in practice it is consistently outperformed by the purely random algorithm in high density and high k value scenarios.