FACULTY OF SCIENCE - SCHOOL OF COMPUTER SCIENCE

PRINCIPLES OF COMPUTER NETWORKS
COMP 3203

DISCOVERY OF ROTATING DIRECTIONAL SENSORS

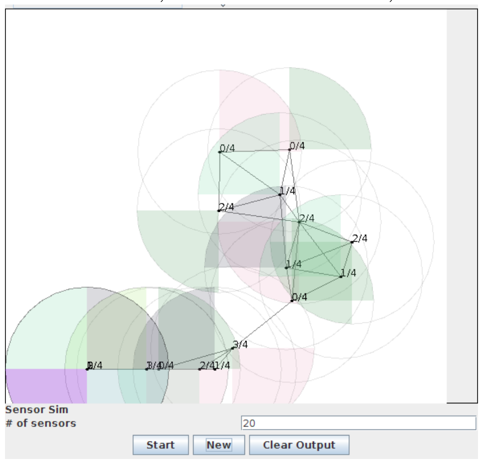
AUTHORS

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

In the ARA 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 must be a prime number based on a colouring of the graph.

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.

1.3 Random Selection Rotation Mechanism Algorithm Prime

RSRMA' operates much the same as RSRMA, except that instead of rotating through the k sectors k times in Mech0, or rotating through k sectors and delaying for k, it passes in a prime number (d) as the second argument. 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 chooses Mech0 or Mech1 at random.

2 Program Breakdown

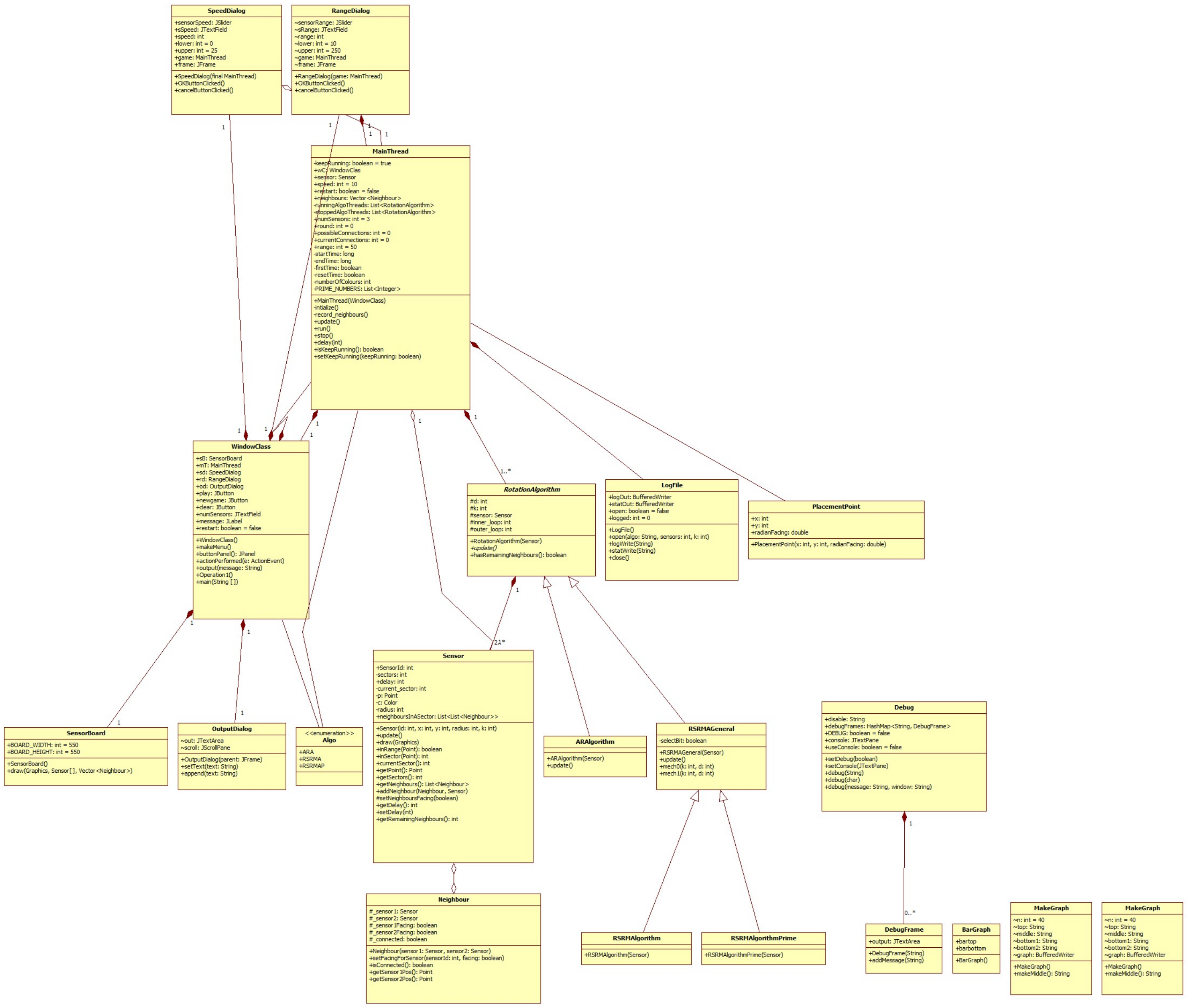
2.1 MainThread

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) «««< HEAD

======= »»»> c85ee15acf9bbaaa36fba9e8326e85f51dcf3bf0 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



3 Comparison Methods

4 Conclusions