The Open University of Israel

Department of Mathematics and Computer Science

Implementation of self-stabilizing algorithms in the distributed model

By

Oleg Zatulovsky

Advanced Project 22997

Prepared under the supervision of Dr. Leonid Barenboim

Table of Contents

Abstract **3**

Introduction **4**

Algorithm 1 ..... **5**

Algorithm 1 Description 5

Algorithm 1 Implementation 6

Algorithm 1 Test 9

Algorithm 2 ..... **11**

Algorithm 2 Description 11

Algorithm 2 Implementation 12

Algorithm 2 Test 16

Algorithm 3 ..... **17**

Algorithm 3 Description 17

Algorithm 3 Implementation 19

Algorithm 3 Test 25

Summary **27**

Installation **30**

Execution **33**

Bibliography **35**

Abstract

A distributed system is defined to be self-stabilizing if, regardless of its initial state, the system reaches a legitimately correct state in a finite time. Graph algorithms and self-stabilization have become one of the most popular fault tolerance approaches and as for today form the basis of many network protocols and clustering tasks. The objective of this project is to implement three self-stabilizing algorithms with different characteristics for graph coloring and compare their performance on various graphs. The results will show us the difference between the theoretical calculations and assumptions made by the authors of the algorithms and the actual performance of the algorithms in real time.

Introduction

In the self-stabilizing distributed model, we consider a partially connected system of autonomous nodes, each of which has limited information about the system. In our model we consider an n-vertex graph $G = (V, E)$ with maximum degree Δ. Each node $v \in V$ hosts a processor with two types of memory. The Read Only Memory (ROM) holds a unique ID number of the node, the degree Δ, the number of vertices n and the algorithm code. The contents of the ROM are initialized once and cannot be altered during execution. The second type of memory is Random Access Memory (RAM). This memory can change during execution and holds variables such as program registers, node color, received data from other nodes etc... The RAM memory is controlled by the user and as a result is prone to faults caused by system crashes and transient faults. We can think of a situation in which as a result of system crash the coloring of the nodes is changed and becomes illegal. Once the faults cease, the self-stabilizing algorithm knows how to self-stabilize the graph into a proper solution.

In this project we will look at three self-stabilizing algorithms which were published in different journals and implement them using Sinalgo platform which is a simulation framework for testing and validating network algorithms. The first algorithm by Gradinariu and Tixeuil (1) is a self-stabilizing randomized coloration algorithm. It works on anonymous networks which means that a unique ID in the ROM is unnecessary. Since the algorithm is probabilistic, we would say that there exists a positive probability under which a legitimate coloring is achieved in finite time of $O((\Delta - 1)\log|V|)$. The second algorithm by Kosowski and Kuszner (2) is supposed to determine a legal coloring of graph G in at most $O(|V||E|^2 \Delta Diam(G))$ moves and assumes that each node is assigned with a unique ID. The third one is a state of the art self-stabilizing $(\Delta + 1)$-coloring algorithm with running time of $O(\Delta + \log^* n)$ which is lower than the Szegedy-Vishwanathan barier of $\Omega(\Delta \log \Delta + \log^* n)$. Presented by Barenboim, Elkin and Goldenberg, the algorithm uses a special variation of Linial's algorithm for $O(\Delta^2)$-coloring and a 2-dementional additive group (AG) algorithm for $O(\Delta^2)$ to $O(\Delta)$ coloring in $O(\Delta)$ rounds.

The algorithms implementation will be followed by comparing their performace on different inputs of veriying size. We will give special attention to the Δ parameter and the number of nodes and edges in the graph. Massage and runtime complexities will be given and compared. Since we are dealing with a realtime perfomance we will measure the local computation time in order to have amore accurate view of the algortihms performace.

Algorithms Description

Algorithm 1 - *M. Gradinariu, S. Tixeuil.*

Self-stabilizing Randomized Vertex Coloration Algorithm (1)

Algorithm 1 Description

This randomized algorithm works on anonymous networks and stabilizes with an unfair scheduler. Each vertex knows the bound Δ on the network degree and maintains an integer $C \in \{0..\Delta\}$ which is its color. Each vertex $i \in V$ knows the colors of its neighbors. Once the color of vertex i is not the largest legal possible (line 1), a fair coin is tossed (line 2) and the color C of vertex i is changed to the largest legal possible integer (line 3). The randomization assures us that there exists a positive probability that only one of the nodes changes its color. As a result, starting from an arbitrary coloring, the system reaches a legal coloring with a positive probability.

Pseudo code

1) **if** $C_i \neq \max\left(\{0..\Delta\} \setminus \cup_{j \in \mathcal{N}_i}\{C_j\}\right)$ **then**
2) **if** $\mathrm{random}(0,1) = 1$ **then**
3) $C_i := \max\left(\{0..\Delta\} \setminus \cup_{j \in \mathcal{N}_i}\{C_j\}\right)$

Complexity

The average number of rounds to achieve legal coloring is given as $\mathrm{O}((\Delta - 1)\log|V|)$.

Algorithm 1 Implementation

The implementation of the algorithm on Sinalgo required rewriting existing classes and methods. The following describes all the involved objects and the logic behind them. To fully understand the explanation, the reader must be familiar with the Sinalgo simulation framework (4).

CustomGlobal.java

Variables

Delta – static integer which holds the calculated maximal rank in the graph.

legalColoring - static Boolean that indicated in each round if the coloring became legal.

colorTable – static Hash table for the coloring of the nodes. It is initialized once and hold the different RGB colors for the GUI representation

The following variables are used for the data acquisition and the generation of the report after the algorithm stabilizes:

startAcq – static Boolean that indicates the exact point when data collection starts.

timer – holds the start time of the stabilization.

runtime – holds the runtime in seconds of the stabilization.

rounds - holds the rounds number of the stabilization.

countMsg – holds the number of massages sent during the stabilization.

Methods

buildHash – this method initialized the hash table for the nodes coloring. Since we use RGB colors, we have 255^3 possibilities. For the colors to be notable on the screen, we divide the range $[100, 255^3]$ into n ranges and pick one color from each range. This way the human eye will be able to distinguish adjacent neighbors more easily.

initDelta – this method returns the highest degree in the graph

preRound – this method is initialized before each round. It starts the process of data acquisition for the final stabilization report and before the first round it initialized the Delta and the hash table for the colors.

postRound – this method is inherited from AbstractCustomGlobal.java and runs after each round. It stops the simulation once the network is stabilized and plots the report of the most recent stabilization to the sidebar output.

GUI buttons

`occurFault` – This method is used to simulate a fault in the network. After the network is stabilized the user can choose which percentage of the nodes will be faulted. Once the faults cease, the user can resume the simulation and the network will stabilized once again into a proper state.

AgreeMsg.java

In each round each vertex tries to agree with its neighbors on the current color. The AgreeMsg includes the nodes color as a meta data.

Variables

`color` – an integer that hold the color of the nodes.

Methods

`toString` – this method plots the nodes color.

`getColor` – this color returns the nodes color.

Node1.java

This class implements the main algorithm. Each node knows its color and delta. In each round each node sends its color to its neighbors and consequently receives their colors.

Variables

`Color` – an integer that represents the color of the node.

`size` – an integer that holds the size of the node on the GUI plot.

`neighbors_colors` – an ArrayList of integers that stores the colors of the neighbors received during the last round.

Methods

`handleMessages` – this method iterates over the received massages and extracts the sent meta data.

`preStep` – this method is trigger before each round. It initializes the neighbor's colors Array and sets the GUI node size.

`init` – this method is triggered once upon node creation. It sets the color of the node to be the ID.

`postStep` – this method is triggered after each round and holds the main stabilization algorithm. If the color is maximal then the node agrees. Otherwise a coin is tossed and with probability of ½ a new color is chosen to be a new maximal color. The nodes color is broadcasted to the neighbors after each round via AgreeMsg.

`getMaximalColor` - this method receives an ArrayList of integers which represent colors. The method returns the maximal color in the ArrayList, meaning the highest value in the range $[0, \Delta]$ and does not exist in the received ArrayList.

`toString` – prints the nodes ID and the color.

`changeColorToNeighbor` – this method is triggered when the user wants to simulate a fault in the system. The nodes color is changes arbitrary to the color of one of its neighbors.

`draw` – this method plots the node to the GUI.

`occur_Error` – a button that simulates an error in a certain percentage of the nodes.

Algorithm 1 Test

In order to test the algorithm, we build different graphs with different topologies and collected the data from the different stabilizations. The testing was done on a home PPC with the following parameters:

Intel Core i5-4460 CPU@3.20Ghz, 4GB RAM, 32-bit OS

The testing included changing the Delta and the number of nodes in the graph. The first simulations started from a mono coloring graph and the last step included generating a fault of 25% of the nodes. When the faults ceased the system stabilized again. The following data was collected from the algorithms testing:

Delta	Stabilization rounds	Massages sent	Total run time [sec]	Average time per round	Number of colors
7	13	3696	0.187	0.014385	6
31	29	57568	0.312	0.010759	18
78	86	426020	1.232	0.014326	42
99	132	1212798	3.385	0.025644	78
At this point we generate 25% fault of the nodes					
99	127	1166508	3.229	0.025425	78

n = 100

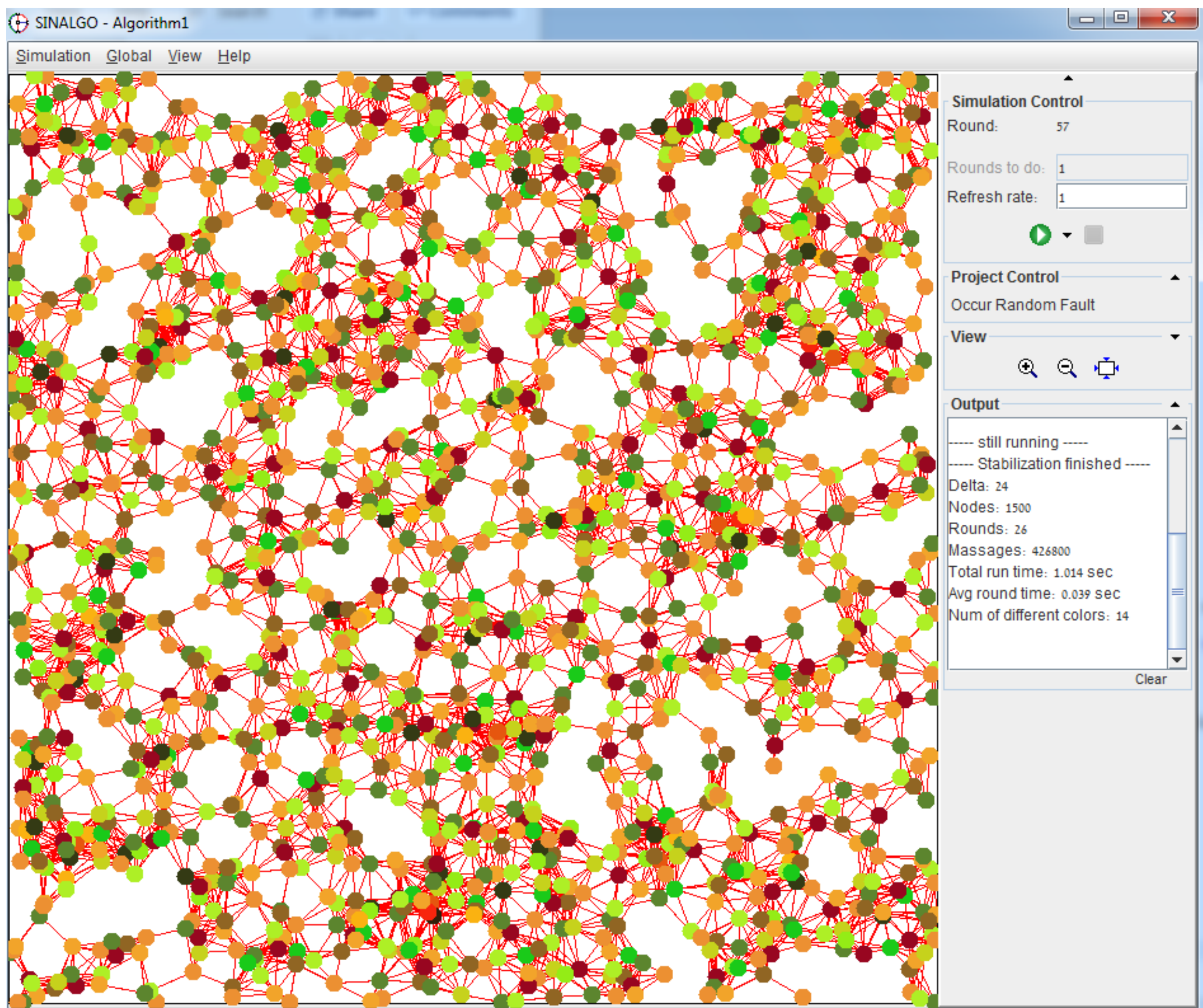
Delta	Stabilization rounds	Massages sent	Total run time [sec]	Average time per round	Number of colors
70	51	3277300	6.848	0.134275	35
131	90	12296062	29.531	0.328122	65
325	171	60082760	205.343994	1.200842	152
609	308	190673402	1051.134033	3.412773	266
At this point we generate 25% fault of the nodes					
609	298	184462542	1032.08606	3.463376	270

n = 1500

Delta	Stabilization rounds	Massages sent	Total run time [sec]	Average time per round	Number of colors
43	43	2859528	6.505	0.151279	23
125	90	23119886	57.938999	0.643767	64
266	139	77231148	297.145996	2.137741	125
318	166	118273980	2298.581055	13.846874	152
At this point we generate 25% fault of the nodes					
318	126	89601500	1894.873047	15.038675	146

n = 3000

An example of a self-stabilized network with 1500 nodes:



1500 nodes Self- Stabilization

Algorithm 2 - *A. Kosowski, L. Kuszner.*

A graph coloring algorithm under a distributed daemon optimal for bipartite graphs (2)

Algorithm 2 Description

In this algorithm, each node is assigned with an ID which. Variable f is used for the construction of a spanning three of the graph what assures us that between every two nodes there will be exactly one path. This algorithm is semi-uniform since just one node is considered as a root. A node $u \in V$ will be assigned as the parent of node $v \in V$ if $f(u) < f(v)$. Lines 1 and 2 construct the spanning tree. Lines 3 and 4 use the mutex $s(u)$ to allow the node *u* to declare that it is interested to change its color $c(u)$. In line 4 the condition on $Id(u)$ is used to prevent parallel color changes on neighboring nodes. In line 6 the parameter $\gamma(u)$ assigns the smallest odd color if the parents color $c(parent(u))$ is even and the smallest even color if the parents color is odd.

Pseudo code

1) **if** $(i \neq root(G))$ and $\left(f(i) \leq \min_{j \in \mathcal{N}_i} f(j)\right)$ **then**
2) $\quad f(i) := \max_{j \in \mathcal{N}_i} f(j) + 1$
3) **if** $(parent(i) \neq null)$ and $(c(i) \neq \gamma(i))$ **then**
4) $\quad s(i) := on$
5) **if** $(s(i) = on)$ and $\left(Id(i) < \min_{j \in \mathcal{N}_i} \{Id(j) \mid s(j) = on\}\right)$ **then**
6) $\quad c(i) \coloneqq \gamma(i)$
7) $\quad s(i) \coloneqq off$

Where $\gamma(i) = \min\{i \in \mathbb{N} \cup \{0\} : A(i) \neq A(c(parent(i)) \; and \; \forall_{u \in N(i)} k \neq c(u)\}$ and $A(i) \equiv i \; mod \; 2$

Complexity

The run time complexity of the algorithm for coloring a graph legally is $O(|E||V|^3 \Delta diam(G))$

Algorithm 2 Implementation

The implementation of the algorithm on Sinalgo required rewriting existing classes and methods. The following describes all the involved objects and the logic behind them. To fully understand the explanation, the reader must be familiar with the Sinalgo simulation framework (4). In order to change the Delta parameter we alter the parameter rMax in the config.xml file.

CustomGlobal.java

Variables

Delta – static integer which holds the calculated maximal rank in the graph.

legalColoring - static Boolean that indicated in each round if the coloring became legal.

colorTable – static Hash table for the coloring of the nodes. It is initialized once and hold the different RGB colors for the GUI representation

The following variables are used for the data acquisition and the generation of the report after the algorithm stabilizes:

startAcq – static Boolean that indicates the exact point when data collection starts.

`timer` – holds the start time of the stabilization.

`runtime` – holds the runtime in seconds of the stabilization.

`rounds` - holds the rounds number of the stabilization.

`countMsg` – holds the number of massages sent during the stabilization.

Methods

`buildHash` – this method initialized the hash table for the nodes coloring. Since we use RGB colors, we have 255^3 possibilities. For the colors to be notable on the screen, we divide the range $[100, 255^3]$ into n ranges and pick one color from each range. This way the human eye will be able to distinguish adjacent neighbors more easily.

`initDelta` – this method returns the highest degree in the graph

`preRound` – this method is initialized before each round. It starts the process of data acquisition for the final stabilization report and before the first round it initialized the Delta and the hash table for the colors.

`postRound` – this method is inherited from AbstractCustomGlobal.java and runs after each round. It stops the simulation once the network is stabilized and plots the report of the most recent stabilization to the sidebar output.

GUI buttons

`occurFault` – This method is used to simulate a fault in the network. After the network is stabilized the user can choose which percentage of the nodes will be faulted. Once the faults cease, the user can resume the simulation and the network will stabilized once again into a proper state.

`show_root` – this option is given for the user to show the root node.

CheckError.java

In each round each vertex sends its neighbors the current color, ID, variable f and the semaphore s. The CheckError class holds the meta data for these pratmeter.

Variables

`senders_color` – an integer the holds the senders color

`senders_ID` – an integer the holds the senders ID

`senders_f` – an integer the holds the sender's parameter f for tree construction.

`senders_s` – a mutex that allows mutual exclusion for color changing in the neighborhood.

Methods

`toString` – this method plots the nodes color.

`getSendersColor` – this method returns the nodes color.

`getSendersID` – this method returns the nodes ID.

`getSendersf` – this method returns the nodes f parameter.

`getSenders_s` – this method returns the nodes mutex state.

Node2.java

This class implements the main algorithm. Each node knows its color and delta. In each round each node calculates gamma, the new parent ID (if applicable) and the new f variable. After each round each node broadcasts these parameters to its neighbors.

Variables

`Color` – an integer that represents the color of the node.

`size` – an integer that holds the size of the node on the GUI plot.

`neighbors_colors` – an ArrayList of integers that stores the colors of the neighbors received during the last round.

`f` - an integer that is used for construction a spanning tree by parent allocation.

`parentID` – an integer that holds the ID of the nodes parent.

`gamma` – an integer that hold the value of gamma as described in the article.

`mutex` – a Boolean that acts as a binary semaphore.

`root` - a Boolean indicator if the current node is the root.

`neighbors_data` – an ArrayList of Quads that stores the data of the neighbors received from the last round.

Methods

`handleMessages` – this method iterates over the received massages and extracts the sent meta data.

`preStep` – this method is trigger before each round. It initializes the neighbor's data ArrayList and sets the GUI node size.

`init` – this method is triggered once upon node creation. It initialized the nodes parameters and by default, set node with ID==1 to be the root.

`postStep` – this method is triggered after each round and holds the main stabilization algorithm. At each step the parent of the node is updated according to the new minimal f value. All the nodes except the root calculate gamma at each round. Algorithmic sections F, C1 and C2 are responsible for the legal coloring and the mutual exclusion.

`toString` – prints the nodes ID and the color.

`changeColorToNeighbor` – this method is triggered when the user wants to simulate a fault in the system. The nodes color is changes arbitrary to the color of one of its neighbors.

`draw` – this method plots the node to the GUI.

`occur_Error` – a button that simulates an error in a certain percentage of the nodes.

`getMinimal_f` – this nethod returns an integer which is the minimal f among the neighbors of this node.

`getMinimal_f_ID` - this nethod returns an integer which is the ID of the minimal f among the neighbors of this node.

`getMaximal_f` – this nethod returns an integer which is the maximal f among the neighbors of this node.

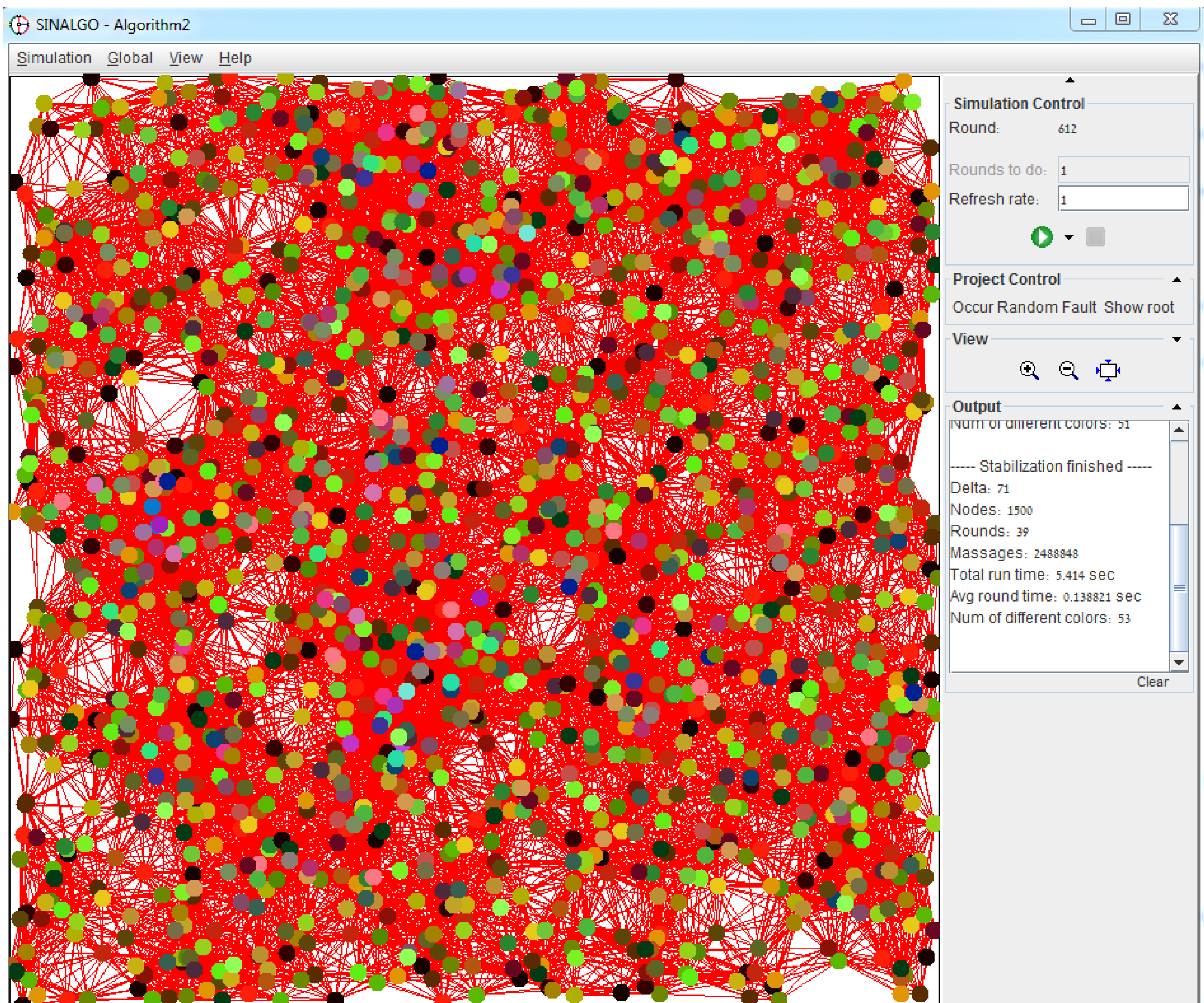
`mutualExclution` - this method returns an integer of the minimal Id among the neighbors of this node with mutex equals TRUE.

`calcGamma` – this method receives the parents color and returns a new value for gamma according to the parent's color.

`noSameNeighbor` – this method receives a color c and returns true if there does not exist a neighbor with color c.

`getParentsColor` – this method extracts the parents color from the received meta data and returns it as integer.

An example of a self-stabilized network with 1500 nodes:



1500 nodes Self- Stabilization

Algorithm 2 Test

In order to test the algorithm, we build different graphs with different topologies and collected the data from the different stabilizations. This algorithm is limited only to connected graphs. Any graph that has more than one connected component will fail to stabilize since the algorithm, as described in the article, is not capable of assigning more than one root node. The testing was done on a home PPC with the following parameters:

Intel Core i5-4460 CPU@3.20Ghz, 4GB RAM, 32-bit OS

The testing included changing the Delta and the number of nodes in the graph. The first simulations started from a mono coloring graph and the last step included generating a fault of 25% of the nodes. When the faults ceased the system stabilized again. The following data was collected from the algorithms testing:

Delta	Stabilization rounds	Massages sent	Total run time [sec]	Average time per round	Number of colors
4	21	7200	0.172	0.00819	3
18	25	33072	0.218	0.00872	14
51	72	240406	0.764	0.010611	40
99	95	857656	2.309	0.024305	53
At this point we generate 25% fault of the nodes					
99	24	209852	0.624	0.026	56

n = 100

Delta	Stabilization rounds	Massages sent	Total run time [sec]	Average time per round	Number of colors
24	96	1581180	3.776	0.039333	20
63	266	17380820	37.236	0.139985	51
124	759	103498836	248.757996	0.327744	92
587	572	353164642	2647.126953	4.627844	384

At this point we generate 25% fault of the nodes					
587	197	121226392	1050.19397	5.330934	384

n = 1500

Delta	Stabilization rounds	Massages sent	Total run time [sec]	Average time per round	Number of colors
41	320	21759628	21759628	21759628	34
97	641	134231040	341.502991	0.532766	75
124	1077	279839624	707.133972	0.656578	89
256	1024	570584388	2746.511963	2.682141	178
At this point we generate 25% fault of the nodes					
256	168	93145252	444.27301	2.644482	179

n = 3000

Algorithm 3 - *Leonid Barenboim, Michael Elkin, Uri Goldenberg*

Fully Dynamic Self-Stabilizing $O(\Delta)$-Coloring (3)

Algorithm 3 Description

This algorithm uses a modified version of Linial's technique for $O(x^2)$ coloring. The heart of the algorithm is based on computing a polynomial $P_u(x)$ for node's u color and for each of u's neighbors. This allow to select a color (for u) such that the color $\langle x, P_u(x)\rangle$ holds $\langle x, P_u(x)\rangle \neq \langle y, P_v(y)\rangle$ for any neighbor v of u. The following definitions are necessary to follow the algorithm. $r = log^* n$ represents the number of iterations in Linial's algorithm until proper coloring is achieved. $t_1 = O(\Delta^2), \dots, t_r = O((\Delta log n)^2)$ represent the upper bounds on the number of colors in the different iterations of Linial's algorithm. Finally, the intervals are used to map each color to an iteration of Linial's algorithm. The intervals are built in such a way that each range holds enough colors to represent all the possible colors for the iterations of Linial's algorithm.

$I_0 = [0, t_1 - 1]$=$[0, \Delta^2 - 1]$ $\qquad$ $\widehat{I_0} = \Delta^2 - 1$

$I_1 = [t_1, t_1 + t_2 - 1]$ $\qquad$ $\widehat{I_1} = t_1 + t_2 - 1 = \Delta^2 + t_2 - 1 = \widehat{I_0} + t_2$

$I_2 = [t_1 + t_2, t_1 + t_2 + t_3 - 1]$ $\qquad$ $\widehat{I_2} = t_1 + t_2 + t_3 - 1 = \widehat{I_1} + t_3$

………………………………………………………………… $\qquad$ …………………………………………………………..

$I_{r-1} = [t_1 + t_2 + \cdots + t_{r-1}, t_1 + t_2 + \cdots + t_r - 1]$ $\qquad$ $\widehat{I_{r-1}} = \widehat{I_{r-2}} + t_r$

$I_r = [t_1 + t_2 + \cdots + t_r, t_1 + t_2 + \cdots + t_r + n - 1]$ $\qquad$ $\widehat{I_r} = \widehat{I_{r-1}} + n$

Each node holds in it's RAM a color and the ROM holds Δ, the number of nodes and the algorithm. In lines 1,2 the algorithm checks if the color is legal, if not the color is changed to an initial state in a high range. In lines 6,7 the algorithm checks if the color range is 2 or higher, in this case the color is updated by modLinial algorithm and a lower range is achieved. In lines 9-11 the color is in the interval I_1 and the node selects a new color from the interval I_0 and distinct from any other adjacent color in I_0. ModLinial selects a new color in I_0 taking in account the set S' of all possible colors that the neighbors (in I_1) can obtain in the current iteration. Finally, in lines 12-16 a color that is in I_0 is finalized or changes to a different color in I_0 according to AG algorithm which provides us with $O\left(\sqrt{\Delta^2}\right) = O(\Delta)$ coloring.

Pseudo code (for node *u*)

1) **if** $(c(u) = c(v) \mid v \in \mathcal{N}_u)$ **then** // the neighbors *u* and *v* have the same color.
2) $c(u) := \sum_{i=1}^{r} t_i + ID(u)$
3) **else**
4) I_j denotes the range of $c(u)$
5) Q denotes the subset of $\{v \mid v \in \mathcal{N}_u \wedge v \in I_j\}$
6) **if** $j \geq 2$ **then**
7) $c(u) :=$mod-Linial$(c(u), Q, \emptyset)$
8) **else if** $j = 1$ **then**
9) $S = \{< a, b > \mid c(u) \equiv < a, b >,\ c(u) \in \cup_{j \in \mathcal{N}_i} \{c(j) \in I_0\}\}$
10) $S' = \{< a, (b + a)\ mod\ q > \mid\ < a, b > \in S\} \cup \{< 0, b > \mid\ < a, b > \in S\}$
11) $c(u) :=$mod-Linial$(c(u), Q, S')$
12) **else if** $j = 0$ **then**
13) **if** $c(u) \equiv < a, b > \in Q$ **then**
14) $c(u) := < a, (b + a)\ mod\ q >$
15) **else**
16) $c(u) := < 0, b >$

Complexity

Given an arbitrary coloring of graph $G = (V, E)$, the algorithm produces a legal $(\Delta + 1)$ – coloring in $O(\Delta + \log^* |V|)$ rounds.

Algorithm 3 Implementation

The implementation of the algorithm on Sinalgo required building new classes and functions in addition to rewriting existing classes and methods. The following describes all the involved objects and the logic behind them. In order to fully understand the explanation, the reader must be familiar with the Sinalgo simulation framework (4). This explanation does not replace the documentation notes in the code.

CustomGlobal.java

CustomGlobal class holds customized global variables and methods for the framework.

Variables

numOfColors – static integer array which holds the number of colors at each iteration of Linial's algorithm. It is a global variable and calculated just once during initialization. It can be known to all nodes since each node can calculate it by itself from n.

sumOfnumOfColor - static integer which holds the initial color of a node subtracted by ID of the node.

interval- static integer array which holds the intervals $I_0, \ldots, I_r$ of the available colors in each iteration.

Delta – static integer which holds the calculated maximal rank in the graph.

q - static integer which holds a prime number in the range [Delta, 2*Delta].

r – static integer which holds the number of rounds in Linial's algorithm.

legalColoring - static Boolean that indicated in each round if the coloring became legal.

colorTable – static Hash table for the coloring of the nodes. It is initialized once and hold the different RGB colors for the GUI representation

The following variables are used for the data acquisition and the generation of the report after the algorithm stabilizes:

`startAcq` – static Boolean that indicates the exact point when data collection starts.

`timer` – holds the start time of the stabilization.

`runtime` – holds the runtime in seconds of the stabilization.

`rounds` - holds the rounds number of the stabilization.

`countMsg` – holds the number of massages sent during the stabilization.

Methods

`isPrime` – receives an integer and returns true if the integer is a prime number and false otherwise.

`postRound` – this method is inherited from AbstractCustomGlobal.java and runs after each round. It stops the simulation once the network is stabilized and plots the report of the most recent stabilization to the sidebar output.

`preRound` – this method is inherited from AbstractCustomGlobal.java and runs before each round. Its first task is to start the data acquisition for the simulation parameters measurement. On the first round we calculate the Delta and consequently the prime number q which will later be used. We make sure that the calculation takes place in the first round of the simulation when all the nodes exist, and the edges are reevaluated.

`buildHash` – this method initialized the hash table for the nodes coloring. Since we use RGB colors, we have 255^3 possibilities. For the colors to be notable on the screen, we divide the range $[100, 255^3]$ into n ranges and pick one color from each range. This way the human eye will be able to distinguish adjacent neighbors more easily.

<u>GUI buttons</u>

`occurFault` – This method is used to simulate a fault in the network. After the network is stabilized the user can choose which percentage of the nodes will be faulted. Once the faults cease, the user can resume the simulation and the network will stabilized once again into a proper state.

<u>Polynomial.java</u>

The stabilization process is using Linial's algorithm for finding distinct colors between neighbors. The color generation is done by finding a unique polynomial for each node color. This class implements a polynomial which in it's turn is used in the modLinial functions. The polynomial's structure is a following:

$p(x) = coeff[0] * x^{degree} + coeff[1] * x^{degree-1} + \cdots + coeff[degree]$

<u>Variables</u>

`coeff` – an integer array which holds the coefficients of the polynomial.

`degree` – an integer which hold the polynomial degree.

<u>Methods</u>

`Polynomial` - A constructor that builds a polynomial from a color and a degree. The coefficients are constructed by changing the current color of the node into base q (q is a prime number) and taking the digits as the coefficients. This way each coefficient will be in the range$[0, q-1]$ and the polynomial will be unique.

`value` – this method receives an integer x and returns the value of the polynomial in x p(x).

`getDegree` – returns the degree of the polynomial.

`toString` – prints the polynomial in the canonical form.

`haveSameValue` – this method receives an array of polynomials and n integer i. The method returns true if one of the polynomials intersects with the current polynomial in point i.

CheckError.java

In each round, every node check if it's state is legal. This is done by sending a massage that holds the nodes color to each one of its neighbors. The class CheckError represents a massage that holds the ID and color of the sending node.

Variables

`senders_color` – holds an integer with the color of the sender.

`senders_ID` - holds an integer with the ID of the sender.

Methods

`CheckError` – A constructor for the massage

`getSendersColor` – returns the color of the sender.

Node3.java

This class implements the main algorithm. Each node knows its color, delta and the number of nodes in the graph. In each round each node sends it's color to its neighbors and consequently receives their colors.

Variables

`Color` – An integer the represents The color of the Node as my_color in the article.

`j` – an integer that represents the range of the color.

`size` – an integer that holds the size of the node on the GUI plot.

`neighbors_colors` – an ArrayList of integers that stores the colors of the neighbors received during the last round.

`Q` – an ArrayList of integers that stores the colors of the neighbors received during the last round and which are in the same range as my Color.

`S_prime` – an ArrayList of < Integer , Integer > which holds the colors of S' as described in the article.

Methods

`getRange` – this method receives a color and returns the range to which the color belongs according to the calculated intervals.

`logStar` – this method receives an integer and returns the result of log*n in base 2.

`log(a,b)` – this method returns the result of $\log_b a$.

`isPrime` – this method receives an integer and returns true if the integer is a prime number,

`loglogn(n,k)` – this method receives two integers n and k and returns the result of $(\log_b \log_b \ldots \log_b n)$, k times.

`haveSameNeighbor` – this method returns true if one of the neighbors has the same color as this node.

`modLinial` – in our algorithm this method is overloaded since it has two implementations which are used in different stages of the stabilization process. The first implementation is used for the rounds in which j > 1. The method receives the nodes color and the neighbors color which are in the same range as the nodes color. For each received color the method generates a unique polynomial. The rank of the polynomial is calculated so that there will be enough polynomials to represent all the potential colors. After the polynomials are ready, we calculate a value for the prime number $\text{q_prime} \in [2\Delta+1, 4\Delta]$. In order to return a new unique color in the correct range I_{j-1}, the method looks for an index i at which the nodes polynomial does not intersect with any other polynomial in its neighborhood. Once found the new color is choses as a concatenation of i and $\mathrm{p(i)}$. If the new color is not in the range of I_{j-1} then the algorithm transforms it into the desired range. Note that in case of two new colors in the same neighborhood which are distant by I_{j-1} from each other, we could end up by having two nodes with the same color. This is easily addressed by returning to the original ID and running the stabilization algorithm for these two nodes. The second implementation is used for the round when j $= 1$. In this round the modLinial function receives in addition to the old color and the neighbors from the same range, a list of forbidden colors which in case of generation can't be returned. The forbidden colors are the colors that the AG algorithm must avoid in order to stabilize into $O(\Delta)$ coloring.

`handleMessages` – this method iterates over the received massages and extract the sent meta data.

`preStep` – this method is triggered before each step. One of the main functions in this implementation is to calculate the color intervals which define the ranges of the colors. This is done before the first

round just after Delta is calculated in CustomGlobal.java and the number of nodes in the graph is finalized.

`init` – this method is triggered once when the node is created and, in our implementation, sets the color.

`postStep` – this method contains the main algorithm as described in the article. Needless to explain the algorithm again. Just to emphasize that at the end of this method we calculate the new colors range, and check if the coloring of the current node is legal. At the end the new color is broadcasted to the neighbors.

`legalColor` – this method returns true if the color of the node is legal per the algorithm's definition.

`toString` – this method prints the parameters of the node.

`changeColorToNeighbor` – this method is triggered when the user wants to simulate a fault in the system. The nodes color is changes arbitrary to the color of one of its neighbors.

`draw` – this method plots the node to the GUI.

`occur_Error` – a button that simulates an error in a certain percentage of the nodes.

`concolor` – this method receives a pair of integers and return an integer of their concatenation.

Algorithm 3 Test

In order to test the algorithm, we build different graphs with different topologies and collect the data from the different stabilizations. The testing was done on a home PPC with the following parameters:

Intel Core i5-4460 CPU@3.20Ghz, 4GB RAM, 32-bit OS

The testing included changing the Delta and the number of nodes in the graph. The first simulations started from a mono coloring graph and the last step included generating a fault of 25% of the nodes. We the faults ceased the system stabilized again. The following data was collected from the algorithms testing:

Delta	Stabilization rounds	Massages sent	Total run time [sec]	Average time per round	Number of colors
21	12	11946	0.171	0.01425	23
33	12	22000	0.188	0.015667	35
77	16	72150	0.344	0.0215	69
99	10	82836	0.39	0.039	98
At this point we generate 25% fault of the nodes					
99	25	217872	0.749	0.02996	98

n = 100

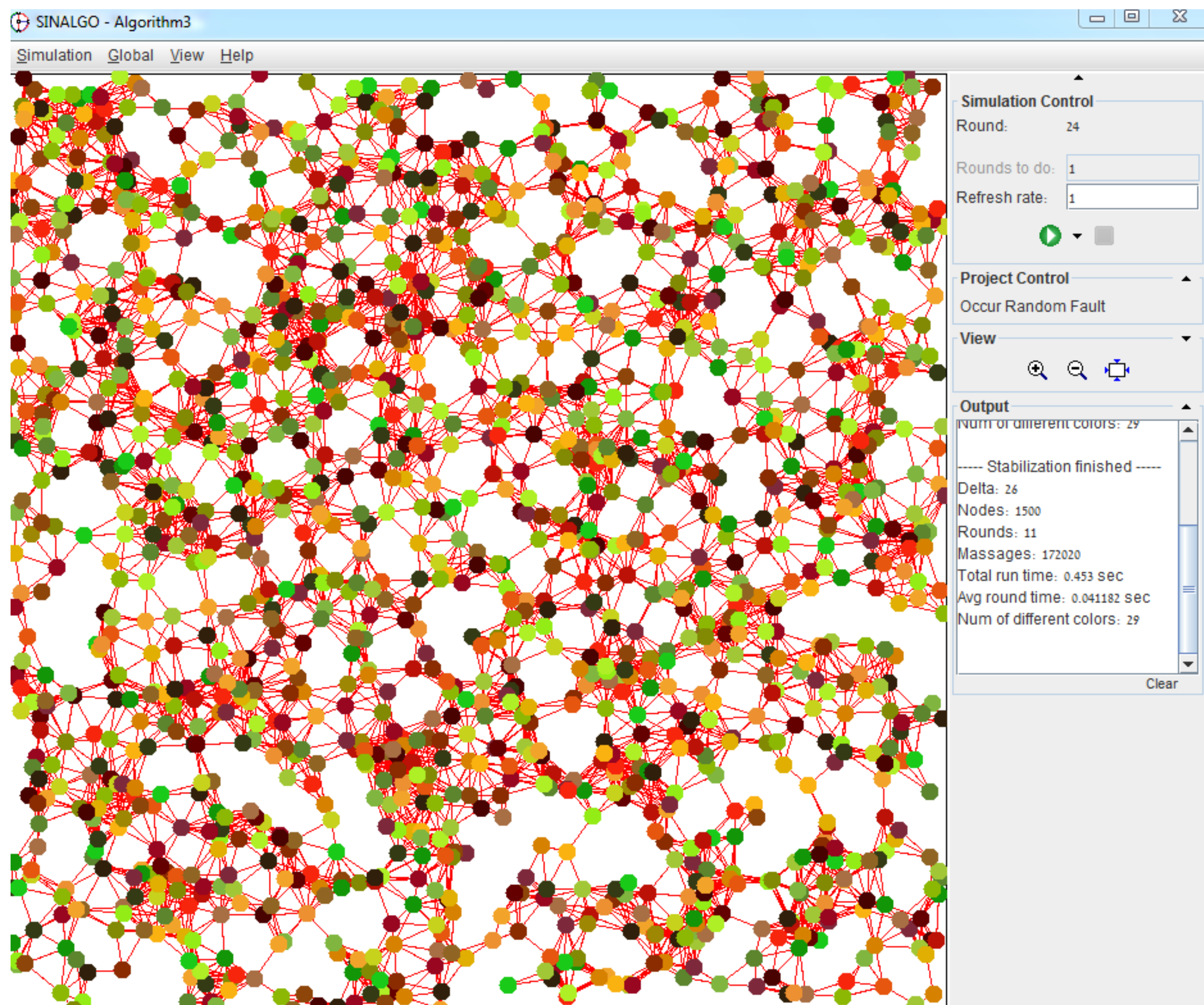
Delta	Stabilization rounds	Massages sent	Total run time [sec]	Average time per round	Number of colors
68	19	1162872	3.088	0.162526	71
214	32	7387982	24.211	0.756594	223
462	20	9422442	55.490002	2.7745	461
601	34	20649618	157.826004	4.641941	586
At this point we generate 25% fault of the nodes					
601	17	10011936	80.995003	4.764412	590

n = 1500

Delta	Stabilization rounds	Massages sent	Total run time [sec]	Average time per round	Number of colors
41	16	1016640	2.933	0.183313	43
85	19	3029796	8.83	0.464737	89
179	27	9620312	31.559	1.168852	181
252	36	19662930	101.744003	2.826222	256
At this point we generate 25% fault of the nodes					
252	47	25842708	130.712997	2.781128	255

n = 3000

An example of a self-stabilized network with 1500 nodes:



1500 nodes Self- Stabilization

Summary

In this project we implemented and tested three self-stabilizing algorithms using Sinalgo platform. According to the authors, the first algorithm by Gradinariu and Tixeuil (1) stabilizes to a legitimate state in $O((\Delta - 1)\log|V|)$ rounds. The following chart represents the results achieved and compared to the theoretical analysis.

Delta Δ	Nodes $\lvert V \rvert$	Stabilization rounds De-Facto	Theoretical analysis $\boldsymbol{O((\Delta - 1)\log\lvert V \rvert)}$	Asymptotical factor C
99	100	127	196	$C = \left\lceil \frac{127}{196} \right\rceil = 1$
325	1500	171	1029	$C = \left\lceil \frac{171}{1029} \right\rceil = 1$
609	1500	298	1931	$C = \left\lceil \frac{298}{1931} \right\rceil = 1$
125	3000	90	431	$C = \left\lceil \frac{90}{431} \right\rceil = 1$
318	3000	126	1102	$C = \left\lceil \frac{126}{1102} \right\rceil = 1$

Algorithm 1 Asymptotical Factor

It is clearly seen that the asymptotical factor is very low in all inputs. We will look at the additional two algorithms and calculate their approximation factor. At the final stage of this summary we will compare the performances of the three algorithms.

Delta Δ	Nodes $\lvert V \rvert$	Edges $\lvert E \rvert$	Stabilization rounds De-Facto	Theoretical analysis $\boldsymbol{O\left(\lvert V\rvert\lvert E\rvert^3\Delta Diam(G)\right)}$ $\boldsymbol{Diam(G)\sim\sqrt{2}\lvert V\rvert}$	Asymptotical factor C
99	100	8932	96	$\Delta <<<$	$C \to 0$
71	1500	64350	189	$\Delta <<<$	$C \to 0$
129	1500	142318	477	$\Delta <<<$	$C \to 0$
263	3000	555766	1032	$\Delta <<<$	$C \to 0$
526	3000	1259634	1568	$\Delta <<<$	$C \to 0$

Algorithm 2 Asymptotical Factor

The factor of $|E|^3$ in the theoretical analysis of algorithm 2 seems to be a very gross bound since the result shows that the self-stabilization of the network in our case is much faster. The following table show the results of Algorithm 3:

Delta Δ	Nodes $\|V\|$	Stabilization rounds De-Facto	Theoretical analysis $O(\Delta + \log^*\|V\|)$	Asymptotical factor C
99	100	25	102	$C = \left\lceil \frac{25}{102} \right\rceil = 1$
214	1500	32	217	$C = \left\lceil \frac{32}{217} \right\rceil = 1$
601	1500	34	604	$C = \left\lceil \frac{34}{604} \right\rceil = 1$
179	3000	27	182	$C = \left\lceil \frac{27}{182} \right\rceil = 1$
252	3000	47	255	$C = \left\lceil \frac{47}{255} \right\rceil = 1$

Algorithm 3 Asymptotical Factor

If we compare the three algorithms, we come to the following conclusions:

	Massages complexity	Result
Algorithm 1	89601500	2
Algorithm 2	93145252	3
Algorithm 3	25842708	1

Massage Complexity

We compare the 3000 nodes test and the self-stabilization after 25% fault and see that Algorithm 3 gives the best result in terms of massage complexity by a factor of 3 from the other algorithms.

	Run Time (Real Time)	Result
Algorithm 1	1894	3
Algorithm 2	444	2
Algorithm 3	130	1

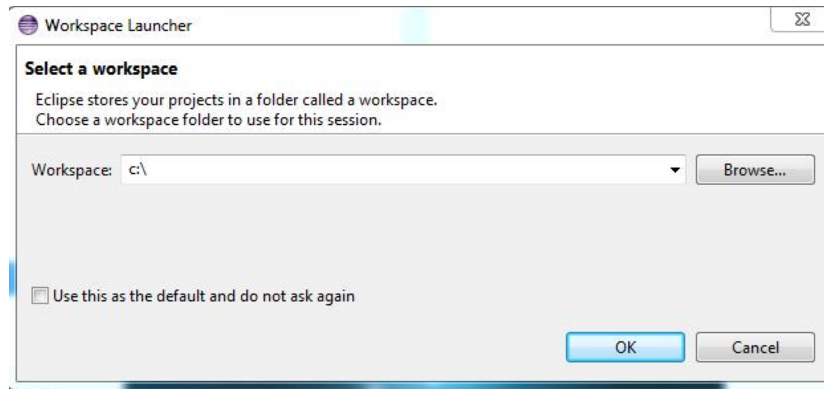
Time Complexity

The real time measurement of the three algorithms shows significant advantage to algorithm 3. We compare the 3000 nodes test and the self-stabilization after 25% fault and see that Algorithm 3 gives the best result in terms of Run Time by a factor of 3 from the second algorithm and a factor of 10 from the randomized algorithm.

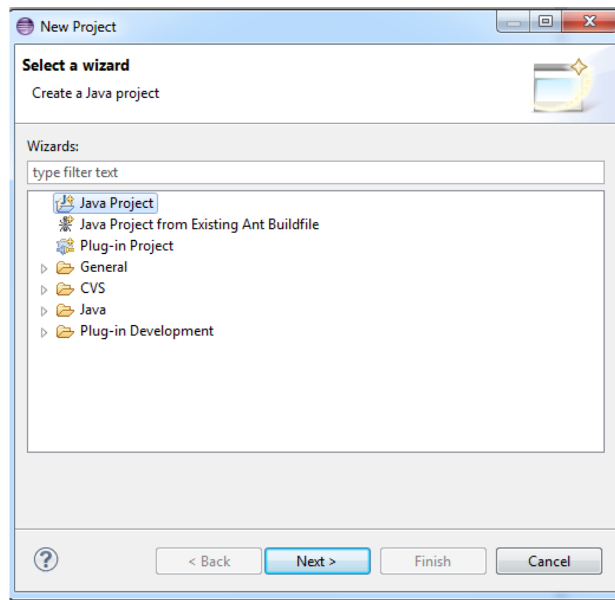
The most important parameter in comparing the algorithms is the stabilization rounds. If we look closely at the asymptotical factor tables, we see clearly that the fastest algorithm by far is algorithm 3.

Installation

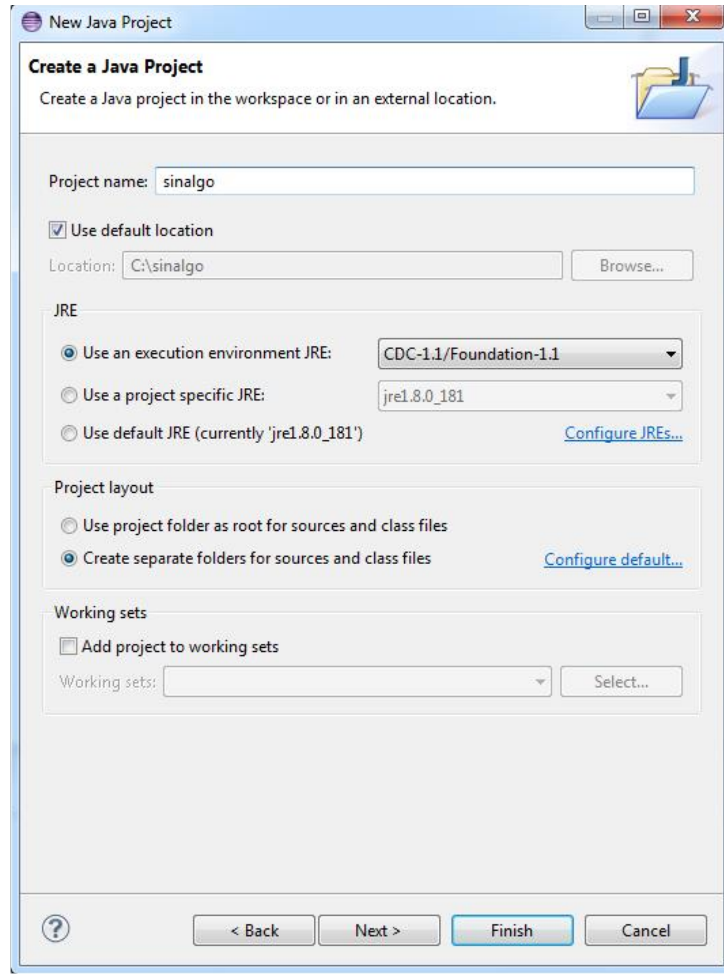
1. **Run Eclipse and select the Workspace c:**



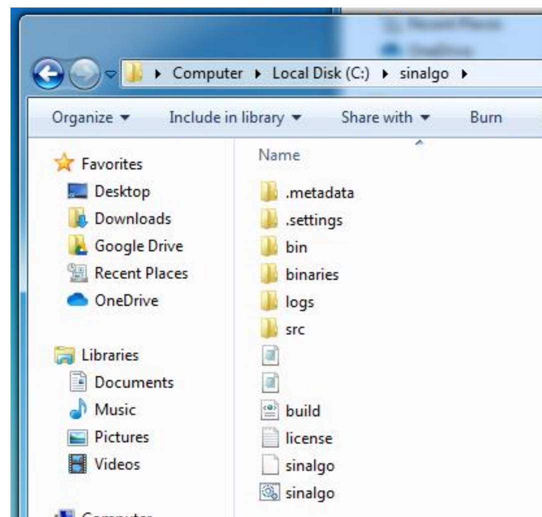
2. **In eclipse create new java project:**



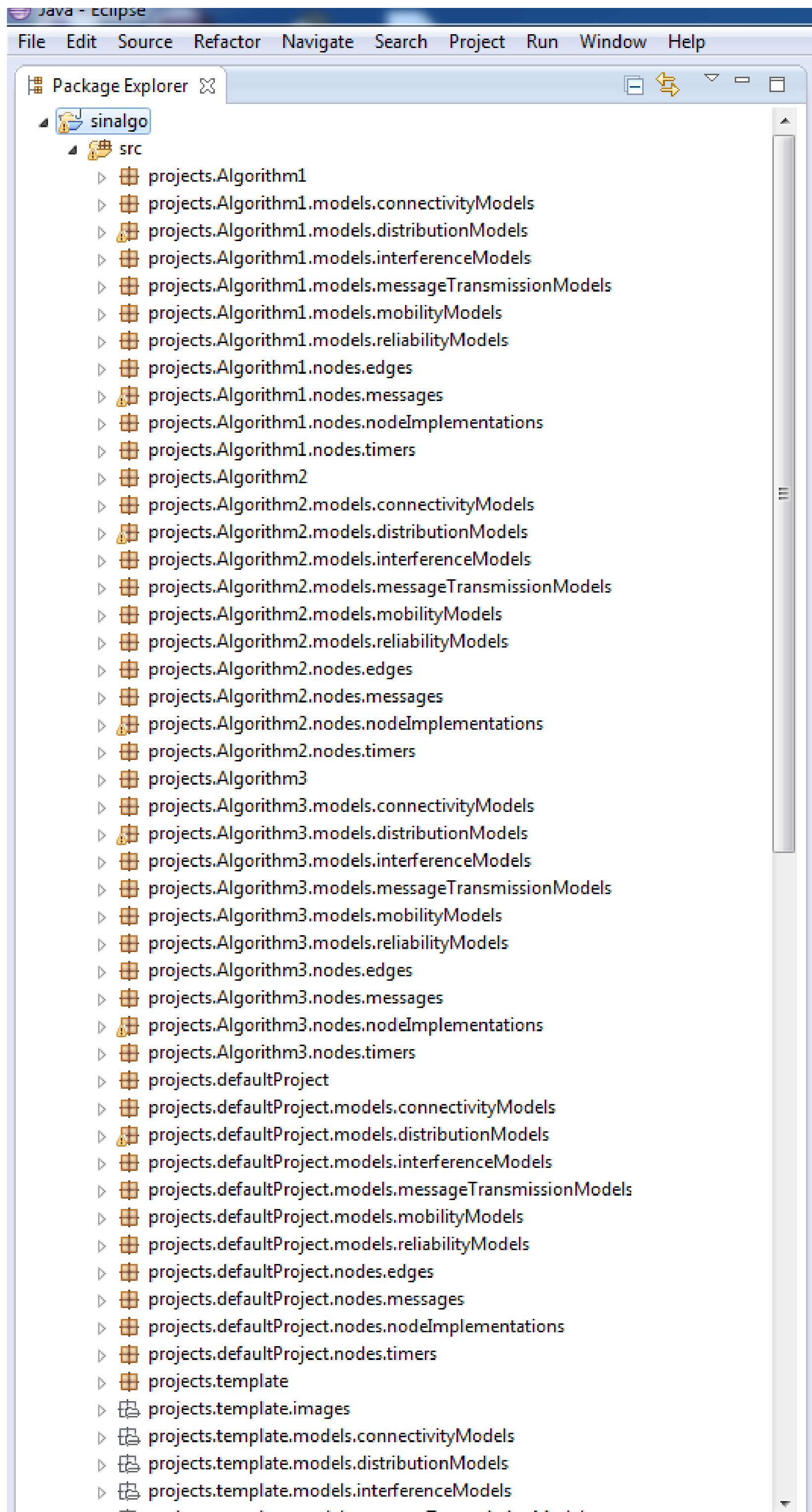
3. **Name the new project as sinalgo:**



4. **<u>Copy the content of the submitted sinalgo folder to c:\sinalgo. The result should be:</u>**

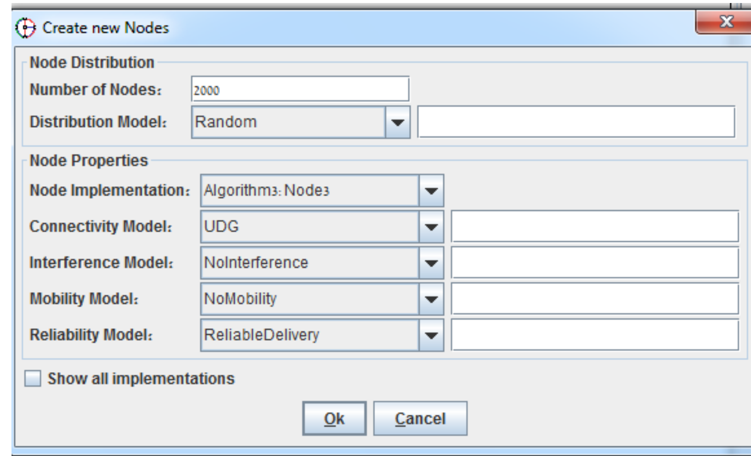


5. In Eclipse the following structure should appear:

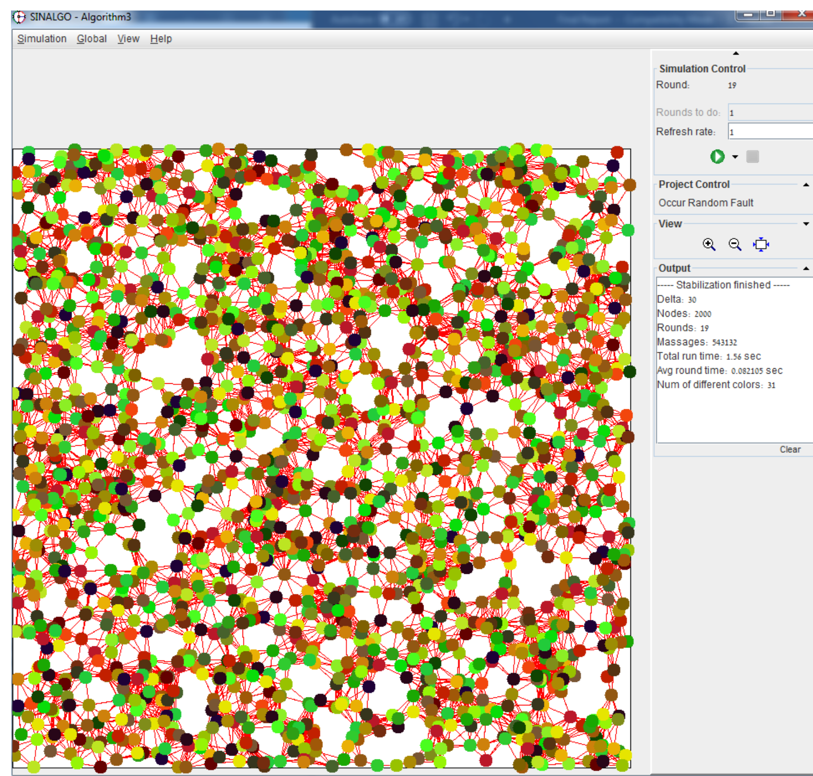


Execution

1. After the Sinalgo simulator and the projects folder have been properly installed, run Sinalgo.
2. Choose Algorithm 1 / Algorithm 2/ Algorithm 3.
3. Create Nodes for the simulation:



4. Run the simulation until it stops with the following report:



5. Click “Occur Random Fault” and choose the percentage of the nodes to fault. It is possible to right click on a specific node and make it fault.
6. Run the simulation again until it stabilizes.

Bibliography

1. *Self-stabilizing vertex coloration and arbitrary graphs.* **M. Gradinariu, S. Tixeuil.** 2000. 4th International Conference on Principles of Distributed Systems. pp. 55-70.
2. *Self-stabilizing algorithms for graph coloring with improved performance guarantees.* **A. Kosowski, L. Kuszner.** 2006, Proc. 8th International Conference on Artificial Intelligence and Soft Computing, p. 1150_1159.
3. *A self-stabilizing algorithm for coloring planar graphs.* **S. Ghosh, M.H. Karaata.** 1993, Distributed Computing, Vol. 7, pp. 55-59.
4. *Self-stabilizing coloration in anonymous planar networks.* **S.T. Huang, S.S. Hung, C.H. Tzeng.** 2005, Information Processing Letter, Vol. 95, pp. 307-312.
5. *Developing self-stabilizing coloring algorithms via systematic randomization.* **S.K. Shukla, D.J. Rosenkrantz, S.S. Ravi.** 1994. 1st International workshop on Parallel Processing. pp. 668-673.
6. *Superstabilizing protocols for dynamic distributed systems.* **S. Dolev, T. Herman.** 1997 , Chicago Journal of Theoretical Computer Science.
7. *Locality in distributed graph algorithms.* **Linial, N.** 1992, SIAM Journal on Computing, Vol. 21, pp. 193-201.
8. *What cannot be computed locally!* **F. Kuhn, T. Moscibroda, R. Wattenhofer.** 2004. 23rd Annual ACM Symposium on Principles of Distributed Computing. pp. 300-309.
9. *A self-stabilizing algorithm for coloring bipartite graphs.* **S. Sur, P.K. Srimani.** 1993, Information Sciences, Vol. 69, pp. 219-227.
10. *Linear time self-stabilizing colorings.* **S.T. Hedetniemi, D.P. Jacobs, P.K. Srimani.** 2003, Information Processing Letters, Vol. 87, pp. 251-255.
11. **GROUP, DISTRIBUTED COMPUTING.** [Online] http://disco.ethz.ch/projects/sinalgo/.
12. **Dolev, Shlomi.** *Self-stabilization.* s.l. : MIT Press, 2000. ISBN 9780262041782.
13. *Locally-Iterative Distributed (Delta + 1)-Coloring below Szegedy-Vishwanathan Barrier, and Applications to Self-Stabilization and to Restricted-Bandwidth Models.* **Leonid Barenboim, Michael Elkin, Uri Goldenberg.** 2017.