# Υλοποίηση Και Οπτικοποίηση Βασικών Αλγορίθμων Για Εκπομπή, Ομαδοποίηση Κόμβων Και Έλεγχο Τοπολογίας Σε Ασύρματα Ad Hoc Δίκτυα

# Implementation And Visualization Of Algorithms For Broadcasting, Clustering And Topology Control In Wireless Ad Hoc Networks

**Διπλωματική Εργασία Του / Diploma Thesis By**
**Γιάννη Αργυρούλη / John Argyroulis**

**Πανεπιστήμιο Θεσσαλίας**
**Τμήμα Ηλεκτρολόγων Μηχανικών Και Μηχανικών Υπολογιστών**
**University Of Thessaly**
**Department Of Electrical And Computer Engineering**

Επιβλέποντες / Supervisors
Δημήτριος Κατσαρός / Dimitrios Katsaros
Παναγιώτης Μποζάνης / Panayiotis Bozanis

Volos 2017, Greece

## Acknowledgements

I would like to thank both professors Dimitrios Katsaros and Panayiotis Bozanis, for giving me the opportunity to work on this project and develop my skills and knowledge in programming web applications. I would also like to thank professor Katsaros for his patience and help with issues regarding the implementation of the algorithms used in the project.

## Περίληψη

Η τεχνολογία μπορεί να αποτελέσει σημαντικό αρωγό στην αύξηση της παραγωγικότητας στους χώρους εργασίας. Το ίδιο συμβαίνει και στην εκπαίδευση. Η τεχνολογία χρησιμοποιείται από δάσκαλους και καθηγητές για να βοηθήσει στην μετάδοση των γνώσεων τους. Χρησιμοποιείται επίσης από τους μαθητές για την αναζήτηση πληροφοριών. Για παραδείγματα ασκήσεων όμως, ο πίνακας αποτελεί το πιο κοινό εργαλείο για τον διδάσκοντα, ενώ για τους μαθητές και φοιτητές το χαρτί και το μολύβι. Στις ασκήσεις των μαθημάτων Ad Hoc Ασυρμάτων Δικτύων, είναι πολύ συνήθης ο σχεδιασμός γραφημάτων δικτύων και τα βήματα εκτέλεσης των αλγορίθμων γράφονται στον πίνακα από τον διδάσκοντα, ενώ οι φοιτητές αντιγράφουν τα γραφήματα και το κείμενο στα χαρτιά τους. Έτσι καταλήγουν να έχουν αφιερώσει τον περισσότερο χρόνο στην αντιγραφή, παρά στην παρακολούθηση των διαλέξεων, ενώ ο καθηγητής συχνά καλείται να αντιμετωπίσει το πρόβλημα έλλειψης χώρου στον πίνακα. Κάποιες φορές η εκτέλεση των αλγορίθμων παρουσιάζεται μέσω διαφανειών, το οποίο απαιτεί μεγάλο αριθμό από αυτές για να περιέχει επαρκείς πληροφορίες, γεγονός που αυξάνει τη δουλειά για την προετοιμασία της διάλεξης. Ο στόχος αυτής της διπλωματικής εργασίας είναι η ανάπτυξη ενός λογισμικού, το οποίο θα διευκολύνει την παρουσίαση και εκμάθηση βασικών αλγορίθμων Ad Hoc Ασυρμάτων Δικτύων, εφαρμόζοντας αυτούς τους αλγόριθμους και παρουσιάζοντας τα αποτελέσματα και τα βήματα εκτέλεσης μέσω γραφικών και κειμένου. Η εφαρμογή μπορεί να χρησιμοποιηθεί από καθηγητές για παρουσιάσεις και από φοιτητές για εξάσκηση στο σπίτι, αντικαθιστώντας την ανάγκη για πίνακα και αντιγραφή στο χαρτί.

**Λέξεις Κλειδιά :** Ad Hoc, Ασύρματα Δίκτυα, Κατανεμημένοι Αλγόριθμοι Δικτύων, Εκπαίδευση, Γράφοι, Γραφήματα, Λογισμικό, Εφαρμογή Ιστού, Node.js, Express.js

## Abstract

Technology can be an important contributing factor for increasing productivity in workplaces. The same applies with education. Technology is used by teachers and professors to aid them in presenting their knowledge. It is also used by students to seek more information. For exercise examples though, the blackboard is usually the professor’s tool, while for the students are the pen and paper. In Ad Hoc Wireless Networks classes’ exercises, it’s very common to draw network graphs and the execution of the algorithms’ steps is written on the blackboard by the professor, while the students copy the graphs and the text in paper. They have to spend most of the time copying than paying attention to the lecture and the professor often has to deal with the blackboard’s limited space. Sometimes the execution of the algorithms is shown in slides, but that usually requires lots of them to adequately present the information, which increases the preparation work for the professor. This thesis’s purpose is to develop a software that will assist teaching and studying basic Ad Hoc Wireless Networks’ algorithms, by implementing these algorithms and presenting the results and the steps of execution with graphics and text. The application can be used for presentations by the professors and practice at home for the students, replacing the need of a blackboard and copying.

**Keywords :** Ad Hoc, Wireless Networks, Distributed Network Algorithms, Education, Graphs, Software, Web Application, Node.js, Express.js

# Table of Contents

Acknowledgements ..... 2
Περίληψη ..... 3
Abstract ..... 3
Chapter 1 – The Application AdHocEd ..... 5
Introduction ..... 5
Algorithms Supported ..... 5
Why A Web Application? ..... 5
Choosing Node.js ..... 5
Software Overview ..... 6
Technical Details ..... 9
How To Install ..... 9
Project Directory Structure ..... 10
Chapter 2 – The Algorithms ..... 11
Introduction ..... 11
Ad Hoc Wireless Networks ..... 11
Terminology ..... 11
Solution Categories ..... 11
The Code ..... 11
Simulation Of Execution ..... 12
Recording Steps Of Execution ..... 12
Wu & Li Algorithm ..... 13
Multipoint Relays ..... 18
DCA (Distributed Clustering Algorithm) Clusters ..... 23
Max-Min D-Cluster Formation ..... 31
Maximal Independent Set (MIS) ..... 35
Local Minimum Spanning Tree (LMST) ..... 40
Relative Neighborhood Graph (RNG) ..... 44
Gabriel Graph (GG) ..... 46
Chapter 3 – Users And Sessions ..... 48
Databases ..... 48
Users ..... 48
Sessions ..... 48
Chapter 4 – Adding More Features ..... 49
Add User Interface Features ..... 49
Add More Algorithms ..... 49
Chapter 5 – References ..... 50
Algorithm Implementation Based On : ..... 50

# Chapter 1 – The Application AdHocEd

## Introduction

This diploma thesis is an educational Web Application, called AdHocEd. The application's purpose is to assist teaching basic Ad Hoc Wireless Networks' algorithms, as a presentation tool for lectures and as a practice tool for students. Professors can use it to show and explain these algorithms' execution steps without having to use the blackboard or slides. Students can pay attention on the lecture instead of spending time copying from the blackboard, since they will be able to run the examples and see the results at any time latter. They can also use the application at home for studying, by creating their own graphs, executing the supported algorithms and seeing the description of the execution process.

The whole project is uploaded here https://github.com/JohnArg/Thesis

## Algorithms Supported

The algorithms supported by the application are :

- Wu & Li Connected Dominating Set
- Multipoint Relays Connected Dominating Set
- DCA clusters
- Max Min D-Cluster formation
- MIS (Maximal Independent Set) Connected Dominating Set
- LMST (Local Minimum Spanning Tree) topology graph
- RNG (Relative Neighborhood Graph) topology graph
- GG (Gabriel Graph) topology graph

These algorithms' implementation is based on the papers mentioned in the **References** section. It should be mentioned that the application does not contain the full description of these algorithms and how they work, only the description of the steps of execution. It is up to the users to study the papers in the **References** and understand how they work before using the software.

## Why A Web Application?

With a Web Application the users don't have to install any software on their machines. They also do not have to worry about hardware requirements to run the application smoothly, since most of the work is done by the server. The web also makes switching to another computer at any time easy and without the cost of reinstallation, since only an internet connection is required.

## Choosing Node.js

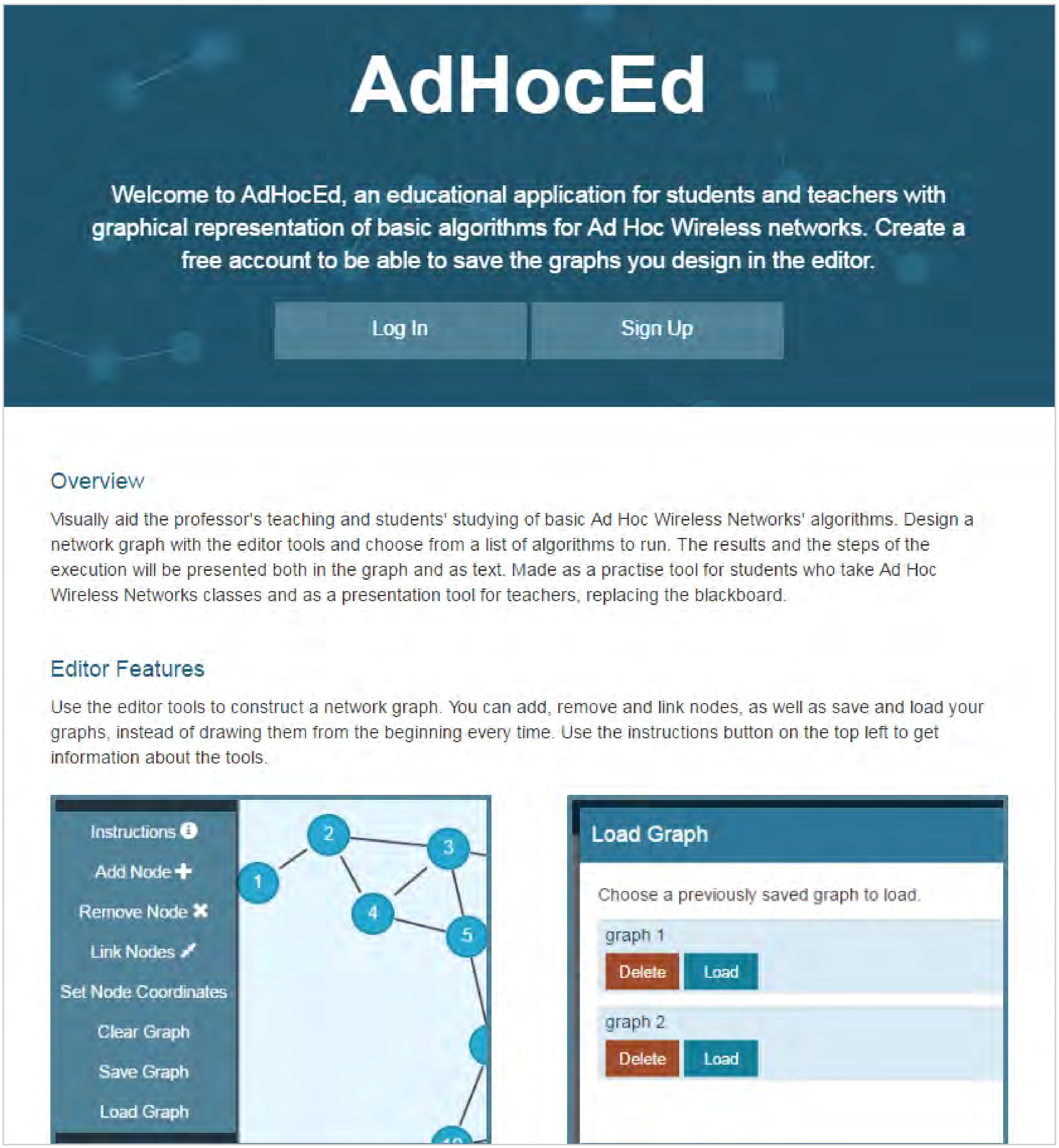
Node.js was chosen for the ability to use JavaScript both on the client and the server code, removing the time cost to learn a new language. It also supports non-blocking I/O operations, which makes it faster in handling such requests. That feature is useful since we use database storage for sessions and user data. It was also chosen for the large list of packages available from npm, which minimize the code and make easier different aspects of the development process. There are packages for creating sessions, database integration, for using templating engines and frameworks, like Express, which is easy and fast to learn so you can start developing your web server as fast as possible. Also, with its module system the code for the project is split into smaller parts, making maintenance and testing easier.

Node.js uses a single threaded model with asynchronous I/O to handle requests, which is not optimal for apps relying on intensive CPU usage. So to take advantage of multiple CPU cores in this application, we used Node's cluster system to run the application on as many instances (workers), as the available CPUs, that share the same port.

## Software Overview

The application is composed of two main web pages, the Home Page and the Workspace. Initially the user is presented with the Home Page, from which they can Log In or create a new account and be redirected to their workspace. The app is also supported without users and sessions, by changing the "sessionsEnabled" variable in /appClobalData.js. This disables saving and loading network graphs and simply uses an "Enter" button in the homepage. The rest of the app, looks and works the same way as with sessions.

**The Home Page** contains small articles explaining the software's purpose and features and Log In/ Sign Up buttons to enter the user's Workspace. The application uses an authentication system with user accounts and sessions, only to support saving and loading saved graphs per user, instead of redrawing regularly used graphs or for preparing for presentations.



AdHocEd

Welcome to AdHocEd, an educational application for students and teachers with graphical representation of basic algorithms for Ad Hoc Wireless networks. Create a free account to be able to save the graphs you design in the editor.

Overview

Visually aid the professor's teaching and students' studying of basic Ad Hoc Wireless Networks' algorithms. Design a network graph with the editor tools and choose from a list of algorithms to run. The results and the steps of the execution will be presented both in the graph and as text. Made as a practise tool for students who take Ad Hoc Wireless Networks classes and as a presentation tool for teachers, replacing the blackboard.

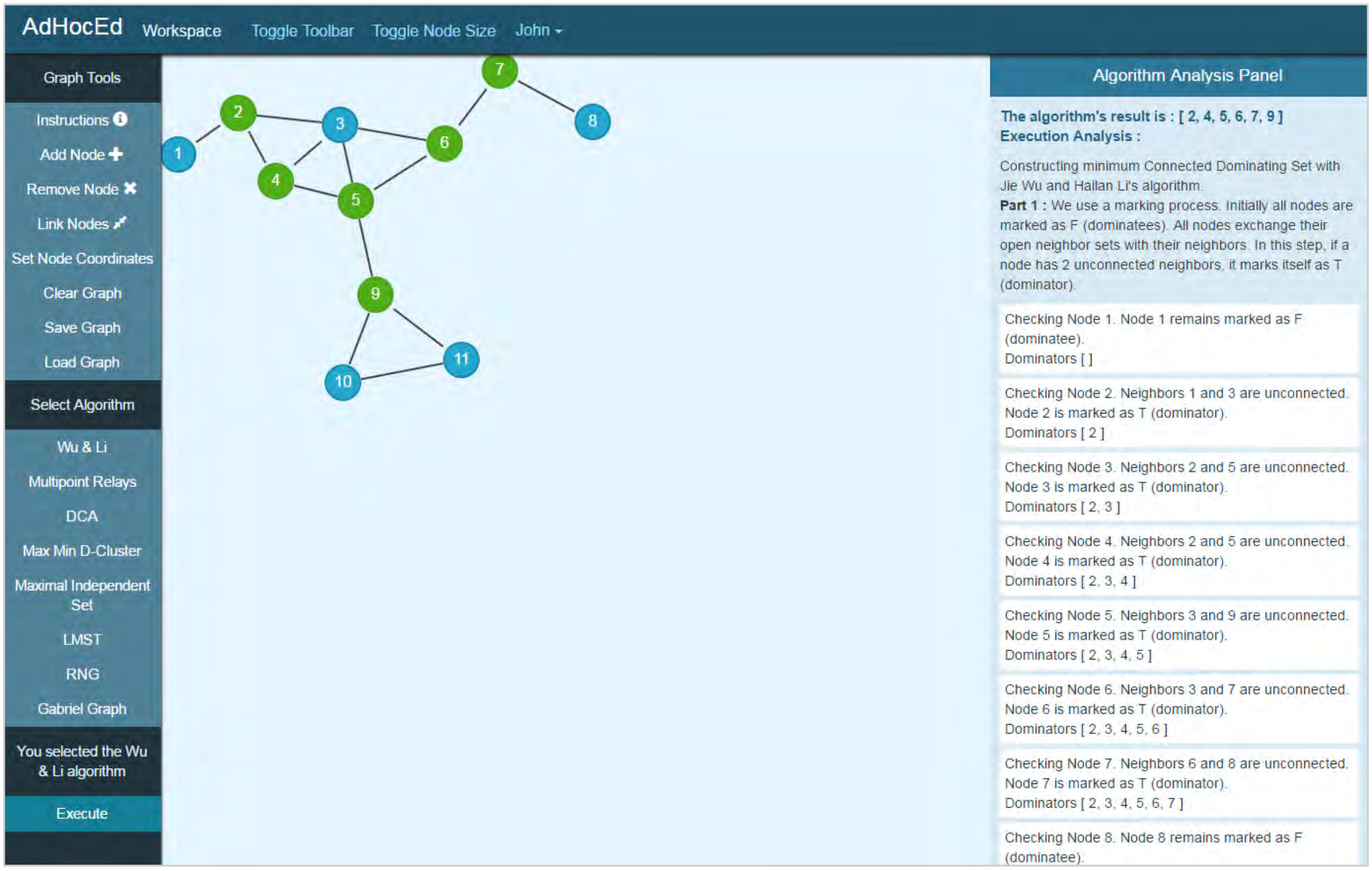
Editor Features

Use the editor tools to construct a network graph. You can add, remove and link nodes, as well as save and load your graphs, instead of drawing them from the beginning every time. Use the instructions button on the top left to get information about the tools.

Load Graph

Choose a previously saved graph to load.

**The Workspace** contains Graph Tools, with which someone can create and edit network graphs, a Graph View where the graphics are drawn, a list of algorithms to execute on the given graph and an Algorithm Analysis Panel, which presents the steps of the execution. It also contains Log Out and Delete Account buttons for the users.



**The algorithm's result is : [ 2, 4, 5, 6, 7, 9 ]**
**Execution Analysis :**

Constructing minimum Connected Dominating Set with Jie Wu and Hailan Li's algorithm.
**Part 1 :** We use a marking process. Initially all nodes are marked as F (dominatees). All nodes exchange their open neighbor sets with their neighbors. In this step, if a node has 2 unconnected neighbors, it marks itself as T (dominator).

Checking Node 1. Node 1 remains marked as F (dominatee).
Dominators [ ]

Checking Node 2. Neighbors 1 and 3 are unconnected. Node 2 is marked as T (dominator).
Dominators [ 2 ]

Checking Node 3. Neighbors 2 and 5 are unconnected. Node 3 is marked as T (dominator).
Dominators [ 2, 3 ]

Checking Node 4. Neighbors 2 and 5 are unconnected. Node 4 is marked as T (dominator).
Dominators [ 2, 3, 4 ]

Checking Node 5. Neighbors 3 and 9 are unconnected. Node 5 is marked as T (dominator).
Dominators [ 2, 3, 4, 5 ]

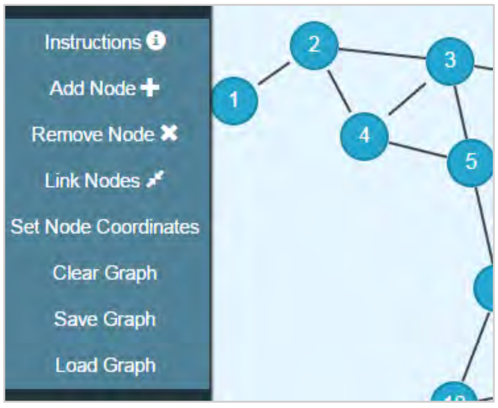
Checking Node 6. Neighbors 3 and 7 are unconnected. Node 6 is marked as T (dominator).
Dominators [ 2, 3, 4, 5, 6 ]

Checking Node 7. Neighbors 6 and 8 are unconnected. Node 7 is marked as T (dominator).
Dominators [ 2, 3, 4, 5, 6, 7 ]

Checking Node 8. Node 8 remains marked as F (dominatee).

## Graph Tools :

They are consisted of an "Instructions" button, which opens a small window that explains the usage of the tools, an add/remove tool to place or remove network nodes, a link tool to connect two nodes, a set node coordinates button to move nodes to a specific position, a clear button to remove everything from the graph view and finally save and load buttons for the ability to reuse graphs.

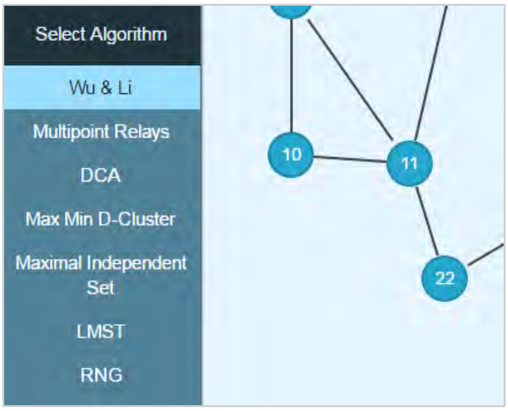


## Graph View :

All of the graphics "reside" in here. The network graphs' vertices (will be referred to as nodes) are represented by colored circles with unique identifications (a positive integer) as text inside the circles and the edges between the vertices by lines (will be referred to as links) connecting the circles. The colors of the nodes and the links change when needed to visually represent an algorithm's result.

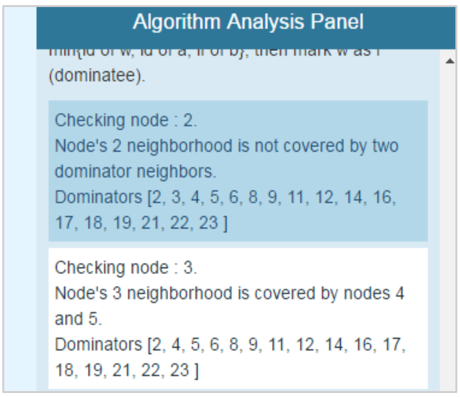
**Algorithm Selection :**

Buttons for choosing one of the supported algorithms and an execution button. In some cases before the execution the user will be asked to give additional data needed as input for the algorithm.



**Algorithm Analysis Panel :**

This panel will contain all the information retrieved by the execution in form of text and "step-boxes". The "step-boxes" will contain explanatory text and results of a particular step of the execution of the selected algorithm. Most of them are clickable and when clicked they change the graphics in the Graph View to properly represent the state of the algorithm for this step.


Algorithm Analysis Panel

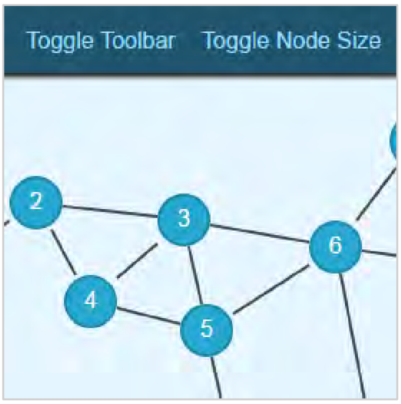
(dominatee).

Checking node : 2.
Node's 2 neighborhood is not covered by two dominator neighbors.
Dominators [2, 3, 4, 5, 6, 8, 9, 11, 12, 14, 16, 17, 18, 19, 21, 22, 23 ]

Checking node : 3.
Node's 3 neighborhood is covered by nodes 4 and 5.
Dominators [2, 4, 5, 6, 8, 9, 11, 12, 14, 16, 17, 18, 19, 21, 22, 23 ]

**Increase Space/Size :**

The "Toggle Toolbar" button in the header can hide the toolbar when additional space is needed for the Analysis Panel's text to be readable. It's very useful with projectors that use a small resolution. The "Toggle Node Size" button will increase the size of the circles representing the network nodes and it's useful when a large graph is resized for a small browser window.



## Technical Details

AdHocEd is a Web Application run on the Node.js platform.

**Client-Side (Front-End) Code Built With:**

- Html
- Css produced with Sass
- JavaScript with jQuery
- Bootstrap
- The Joint.js API with a few modifications to create the graphics for the network graphs in the Graph View.
- Handlebars as the templating engine to dynamically create elements on the pages

**Server-Side (Back-End) Code Built With:**

- Node.js platform (JavaScript run on the server)
- Express.js framework
- Handlebars as the templating engine to render the web pages and send them to the client
- Undescore.js package for its list and object manipulating functions
- MySQL for storing user data and sessions in databases

## How To Install

### Set Up Main App

Download the source code from https://github.com/JohnArg/Thesis. To run the AdHocEd app, one must install the Node.js platform first and then use the npm package manager included in Node's set up, to install the project's dependencies, (declared in the package.json file). To install all the dependencies automatically, simply go the project's folder and type : npm install.

### Set Up The Databases

The app supports two options :

1) Users and sessions, for the ability to save/load graphs (by default on)
2) Free entering without sessions, without saving/loading graphs

For the first option the databases for the user data and the sessions require the installation of MySQL on the machine. To create the two databases, run the /Database/create_db.sql script with MySQL. The table where the sessions are stored is automatically created when the application starts from within the app, if it doesn't already exist. To change the settings of the connection to these two databases, edit the dbConfig object in /Database/queries.js for the users and the sessionStoreConfig object in /Database/sessions.js for the sessions. Also, the users in the configuration objects must be manually created and given the right privileges in mysql, so the app can connect to the databases.

### Run

Before running make sure to change the server's url to your machine's one, in /Client/JavaScript/server_url.js

The command to run the app is :

node adHocEd.js

(or for some systems : nodejs adHocEd.js ).

The application runs by default in http://localhost:3000 unless the constant "port" is changed in /adHocEd.js

## Project Directory Structure

Some of the files and folders included in the app have trivial importance to be mentioned in this section of the paper.

**\adHocEd.js**
The main application file, which is passed as a parameter to the node command to run the app. Starts a server and listens on the specified port for connections. Serves static files to the client and uses the homePageRouter and the workspaceRouter modules in \Server\Routers to route client requests.

**\appGlobalData.js**
Contains global application data that can be accessed by any module with the "require" command.

**\Client :**
All the client related code is inside this folder.

- **\Css :** contains scss files and the compiled css files requested by the client
  - **\External :** css files used by JavaScript extensions
- **\Fonts :** contains glyphicons requested by Bootstrap
- **\Images :** the images of the Home Page
- **\JavaScript :** the JavaScript scripts that handle the view of the pages, the Ajax requests to the server and the data and visualization of the network graphs designed in Graph View
  - **\External :** JavaScript files related with extensions used in the project and their dependencies
- **\Views :** .hbs files that are rendered to HTML by the Handlebars Templating Engine

**\Database :** files related with database creation and connection, query execution and session management.

**\Server :** all the server-work related files.

**\Algorithms :** Each of these files except from steps.js and networkOperations.js is a JavaScript module used to run a specific algorithm. The steps.js file is a module used inside the algorithm modules to record each step's execution data and send that to the client for visualization of the results. The networkOperations.js file is a module containing functions that retrieve data from a network object and are used very often within the algorithm modules' code.

**\Routers :**

- The homePageRouter module is responsible for routing user requests that are relevant to the Home Page. This includes the Log in and the Sign Up requests.
- The workspaceRouter module handles all requests sent from a user's workspace. Such requests could be executing an algorithm passing the request's data, saving, deleting or loading a network graph, logging out of the system or deleting the user's account.

# Chapter 2 – The Algorithms

## Introduction

In this Chapter, we are going to explain how the most important parts of each algorithm's code work, by presenting snippets of the code and the usage of some variables and functions in the solution. As mentioned above, the implementation is based on the papers in References.

## Ad Hoc Wireless Networks

In such temporary networks with mobile nodes, there is no centralized administration infrastructure to organize communications between the nodes. The latter are distinguished by unique identifications (referred to as IDs from now on), can only detect and exchange messages with other nodes within their transmission range and have no knowledge of the rest of the network topology. The algorithms implemented here, are all distributed, meaning that each node executes them to make decisions about his role in the network and how to efficiently communicate with the other nodes.

## Terminology

Given a connected graph G = (V,E) that represents an Ad Hoc Wireless network, the vertices V are called nodes and represent the devices that communicate wirelessly and the edges E (sometimes mentioned as links) represent that the two nodes connected can reach and communicate with each other. The vertices belonging to an edge will sometimes be differentiated as the source and the target. Uni-directional links point from the source to the target with an arrow and define only one direction of communication, while in bi-directional links, the communication can happen from both directions. All the nodes that can be reached from a node A with its maximal transmission power, are called the (1-hop) neighborhood of A. The neighbors of all the 1-hop neighbors of A, that are not directly connected to A, are the 2-hop neighborhood of A and so on.

## Solution Categories

The solutions that the following algorithms provide are divided in three categories :

- Connected Dominating Set (CDS) : A dominating set of  a network graph G, is a subset D of the vertices of G, such that every vertex of G is in either D, or adjacent to a vertex of D. A dominating set is connected when D is a connected subgraph induced by G. The nodes of D are called dominators and all nodes of the network become either dominators or dominatees. The dominators are those that rebroadcast incoming messages and form routing paths. The dominatees use the dominator subnetwork to retransmit their messages.
- Clustering : In Ad Hoc Wireless Networks clustering is crucial for controlling the spatial reuse of the shared channel (for time/frequency division) and for minimizing the amount of data to be exchanged. The clusters are consisted of a clusterhead and some ordinary nodes. Clusterheads are the coordinators of the clusters and act as temporary base stations, since the network is mobile and its topology changes. Gateways are the ordinary nodes that can communicate with nodes belonging to another cluster.
- Topology Control : One of the most important issues in wireless multi-hop networks is to determine transmission power of each node, so as to maintain network connectivity while consuming the minimum possible power. This affects network spatial reuse and traffic carrying capacity. It also has an impact on the battery life of the network nodes, which is a critical resource. The nodes can use systems like GPS to calculate their coordinates and their distances with their neighbors, to be able to use a topology algorithm.

## The Code

Since the project code is comprised of many files, some used from external sources, it cannot be presented fully in this paper.

The whole project can be found here : https://github.com/JohnArg/Thesis

In the algorithms, some important code snippets will be included, to provide the reader with an explanation of the thought process in that part of the application.

## Simulation Of Execution

The software uses the network object sent by the client, which maintains information about each node, such as its identification, its position in the x and y axis in the client's Graph View and its 1-hop neighborhood. It then implements the algorithms based on this knowledge.

It's not the software's purpose to realistically simulate communication conditions of a wireless network between the nodes. For instance, in some occasions that several nodes are supposed to broadcast messages at a specific time step, priority is either given with a message queue or by iterating through a list. This happens because in the code, we iterate through the nodes of the network and for each iteration, if needed, we add a message to be send in the next step to a queue or transmit in the current step. That creates the order of messaging.

Also, at some occasions, nodes are supposed to exchange information about their neighborhood with each other. These messages are not simulated by the software, since the data already exists in the neighbors list inside each node object. It only adds extra data and fills the Analysis Panel in the user interface with useless information, since the client can refer to the graph in the Graph View to see the neighborhood of each node.

## Recording Steps Of Execution

Before any of the algorithms are presented, the process of recording the steps of the execution will be described.

At the server side code, we use the module /Server/Algorithms/steps.js. This module uses two main JavaScript classes, the Step class and the Solution class.

### Step Class

It will hold information regarding a particular step of the execution. It is declared as :

```
var Step = function(){
    this.text = "";
    this.data = {};
}
```

Objects from this class will contain a text for the description of the step of execution and a data object containing the state of the algorithm's results at that time.

### Solution Class

An instance of this class is used to bundle together steps that refer to a particular part of an algorithm. Many of the algorithms' solutions are comprised of different parts, each of which solves a specific problem before moving to the next part. A Solution object is created for each part, to hold the information regarding only that state of the execution.

```
var Solution = function(){
    this.text = "";
    this.steps = [];
    var stepList = this.steps;
    this.result = {};
    this.createStep = function(){
        var step = new Step();
        stepList.push(step);
    }
}
```

The object contains a text variable, used to write a description for that particular part of the execution. The steps array will hold all the Step objects created until the solution is finished. The stepList simply points to the steps array, because due to JavaScript features, the createStep function uses its own "this" object and can't read the this.steps variable. The result variable will contain the result at the end of the solution. The createStep function is used to automatically create a Step object and push it to the stepList. Note that not all algorithms need to use all the variables of a Solution object.

## Wu & Li Algorithm

It is implemented by the /Server/Algorithms/wu_li_cds.js module. The algorithm's purpose is to calculate a Minimum Connected Dominating Set for efficient routing, on a wireless Ad Hoc network. After the execution, the nodes know if they are a Dominator or a Dominatee.

The algorithm is consisted of 3 parts. The first part is the marking process. The second part is Rule 1 and the third Rule 2, both of them reducing the number of dominators. All parts will be explained in the following sections along with their respective code snippets.

### Main Class

The Wu_Li_CDS class creates objects that contain three solutions, each for every part of the execution. The final result is actually the result stored in the "rule2" solution. The main function used to start the calculations is calculateWuLi.

```
//Main object to be returned
var Wu_Li_CDS = function(){
    var that = this;
    that.step1Solution = solutionFactory.newSolution();
    that.rule1Solution = solutionFactory.newSolution();
    that.rule2Solution = solutionFactory.newSolution();
    that.solution = { //This object will contain a solution object for each of the 3 parts of the algorithm
        "step1" : that.step1Solution,
        "rule1" : that.rule1Solution,
        "rule2" : that.rule2Solution
    };

    //This function will use the Wu & Li algorithm to find a minimum CDS
    that.calculateWuLi = function(network){
        _covertToWuLiNodes(network);
        that.solution["step1"].result["dominators"] = _implementWLStep1(network, that.solution);
        that.solution["rule1"].result["dominators"] = _implementWLRule1(network,
that.solution["step1"].result["dominators"], that.solution);
        that.solution["rule2"].result["dominators"] = _implementWLRule2(network,
that.solution["rule1"].result["dominators"], that.solution);
        return that.solution;
    };
}
```

## Marking Process

The algorithm states that initially all nodes are marked as F (dominatees). All nodes exchange their open neighbor sets with their neighbors. In this step, if a node has 2 unconnected neighbors, it marks itself as T (dominator).

Before starting the marking process we use the _covertToWuLiNodes function to add an extra property to each node object of the network. The property is a Boolean variable “dominator”, initialized to false. The function _implementWLStep1 is used to implement the marking process. It takes as a parameter the network object sent by the client and the solution object referring to “step1”, to record the steps of execution in it.

It iterates through each node of the network and checks its neighbors. For every neighbor, it uses a second loop to check if it is connected with the rest of the neighbors. If it isn’t, then it breaks out of that loop and adds the node to a dominators list.

```
//Implements first step of the Wu&Li algorithm
var _implementWLStep1 = function(network, solution){
    var dominatorList = [];
    solution["step1"].text = "Constructing minimum Connected Dominating Set with Jie Wu and Hailan Li's algorithm.<br/>\
    <strong>Part 1 :</strong> We use a marking process. Initially all nodes are marked as F (dominatees).\
    All nodes exchange their open neighbor sets with their neighbors. In this step, if a node has 2 unconnected neighbors,\
    it marks itself as T (dominator). <br/>";
    //Initial decision without Rule1 && Rule 2 ========
    //for every node
    for(var i=0; i<network.nodes.length; i++){
        let neighborsConnected = true;
        //for every neighbor of that node
        solution["step1"].createStep();
        solution["step1"].steps[i].text = "Checking Node " + network.nodes[i].id + ".";
        for(var j=0; j<network.nodes[i].neighbors.length; j++){
            let tempNode = network.nodes[netOperator.returnNodeIndexById(network.nodes[i].neighbors[j], network)];
            let neighborCheckList = network.nodes[i].neighbors;
            for(var k=0; k<neighborCheckList.length; k++){
                if(neighborCheckList[k] != tempNode.id){     //skip checking my own id
                    if(!_hasNeighbor(tempNode, neighborCheckList[k])){
                        neighborsConnected = false;
                        solution["step1"].steps[i].text += " Neighbors "+tempNode.id+" and "+neighborCheckList[k]+" are
unconnected.</br>";
                        break;
                    }
                }
            }
            if(!neighborsConnected){
                dominatorList.push(network.nodes[i].id);
                network.nodes[i].dominator = true;
                solution["step1"].steps[i].text += " Node "+network.nodes[i].id+" is marked as T (dominator).";
                solution["step1"].steps[i].data = { "dominators" : dominatorList.slice()};
                break; //no need to check the other neighbors
            }
        }
        if(neighborsConnected){
            solution["step1"].steps[i].text += " Node "+network.nodes[i].id+" remains marked as F (dominatee).";
            solution["step1"].steps[i].data = { "dominators" : dominatorList.slice()};
        }
    }
    return dominatorList;
}
```

## Rule 1

The paper, in the References section, states that we consider the connected graph of the dominators that were calculated in the Marking Process. If there is a node A and a node B, such as node A’s closed neighborhood set (including A) is a subset of node B’s closed neighborhood set, then if the id of node A is smaller than the id of node B, A is marked as a dominatee. It is implied that A and B are connected. This eliminates some of the dominators calculated in the Marking Process that are not needed, since another node with a higher id covers their neighborhood.

The code uses the _implementWLRule1 function for this job. If there are more than one dominators in the dominator list from the previous process, we iterate through those dominators. For each dominator, let’s say A, we take only its dominator neighbors with higher ids. Iterating through the checkNodeList, we check if any of the candidates covers the neighborhood of A, meaning if A’s neighborhood is a subset of the candidate’s neighborhood. If at least one does, A is marked as dominatee, otherwise it remains a dominator.

```
//Implements the Rule 1 of the algorithm
var _implementWLRule1 = function(network, dominatorList, solution){
    var curNode;
    var checkNodeList;
    var otherDom;
    var reducedNeighborSet;
    var newDominatorList = dominatorList;
    solution["rule1"].text = "<strong>Rule 1 :</strong> Consider 2 dominator nodes <strong>a</strong> and
<strong>b</strong>. \
    If the neighborhood of <strong>a</strong> is a subset of the neighborhood of <strong>b</strong>\
    and id of node <strong>a</strong> < id of <strong>b</strong>, mark <strong>a</strong> as F (dominatee). It is implied
that <strong>a</strong>\
     and <strong>b</strong> are connected.";
    //Traverse the list of the dominators
    if(dominatorList.length > 1){
        for( var p=0; p<dominatorList.length; p++){
            solution["rule1"].createStep();
            //get the dominator node object
            let index = netOperator.returnNodeIndexById(dominatorList[p], network);
            curNode = network.nodes[index];
            solution["rule1"].steps[p].text = "Checking node "+dominatorList[p]+".</br>";
            //Rule 1 canditates are only the 1-hop dominator neighbors with higher ids
            checkNodeList = curNode.neighbors.filter(function(elem) {
                return (network.nodes[netOperator.returnNodeIndexById(elem, network)].dominator)&&(elem>curNode.id);
            });
            if(checkNodeList.length == 0){
                solution["rule1"].steps[p].text = "No other dominator covers the neighborhood of "+curNode.id+".</br>";
                solution["rule1"].steps[p].text += "Node "+curNode.id+" remains a dominator.";
                solution["rule1"].steps[p].data = { "dominators" : newDominatorList.slice()};
                continue;
            }
            //for every other dominator "otherDom" of that list, check if the neighbors of curNode are a
            //subset of the neighbors of otherDom
            let success = false;
            for(var d=0; d<checkNodeList.length; d++){
                otherDom = network.nodes[netOperator.returnNodeIndexById(checkNodeList[d], network)];
                /*curNode will have otherDom as neighbor but of course otherDom won't have itself.
                So we use a temporaty list of curNode's neighbors without otherDom, for subset comparison.
                */
                reducedNeighborSet = curNode.neighbors.filter(function(index) {
                    return index != checkNodeList[d];
                });
                if(_isSubsetOf(reducedNeighborSet, otherDom.neighbors)){
                    success = true;
                    break;
                }
            //if another node does cover the neighborhood of this dominator
            if(success){
                //the new list won't contain this node
                newDominatorList = newDominatorList.filter(function(el){
                    return el != curNode.id;
                });
                solution["rule1"].steps[p].text = "Dominator neighbor "+otherDom.id+" covers the neighborhood of
"+curNode.id+".</br>";
                solution["rule1"].steps[p].text += "Node "+curNode.id+" is marked as F (dominatee).";
                curNode.dominator = false;
                solution["rule1"].steps[p].data = { "dominators" : newDominatorList.slice()};
            }
            else{
                solution["rule1"].steps[p].text = "No other dominator covers the neighborhood of "+curNode.id+".</br>";
                solution["rule1"].steps[p].text += "Node "+curNode.id+" remains a dominator.";
                solution["rule1"].steps[p].data = { "dominators" : newDominatorList.slice()};
            }
        }
    }
    else{
        solution["rule1"].createStep();
        solution["rule1"].steps[0].text = "No further computations needed.";
        solution["rule1"].steps[0].data = { "dominators" : newDominatorList.slice()};
    }
    return newDominatorList;
}
```

**Rule 2**

Rule 2 reduces even further the dominators. If there is a dominator A that has two dominator neighbors B and C and the neighborhood of A is a subset of the union of the neighborhoods of B and C, then if A has the smallest id between those three, A can be marked as dominatee. The Rule implies that B and C are connected. For implementing this rule, we use the domNeighbors array in the code, to keep the dominator neighbors of a dominator A and for every two of them, we check if the union set of their neighborhoods can cover the neighborhood of A. If so the node is excluded from the dominator list returned in the end.

```
//Implements Rule 2
var _implementWLRule2 = function(network, dominatorList, solution){
     var curDom;
     var domNeighbors = [];
     var unionSet;
     var success; //used for deciding the appropriate text for a step
     var newDominatorList = dominatorList;
     solution["rule2"].text = "<strong>Rule 2 :</strong> Consider 2 dominator nodes <strong>a</strong> and
<strong>b</strong> that are both neighbors\
      of another dominator <strong>w</strong>. If the neihborhood of <strong>w</strong> is a subset of the union of the
neighborhoods of <strong>a</strong>\
       and <strong>b</strong> and id of w = min{id of w, id of a, if of b}, then mark w as F (dominatee). It is implied that
<strong>a</strong>\
      and <strong>b</strong> are connected.";
     if(dominatorList.length > 1){
          //Traverse the dominators list
          for(var g=0; g< dominatorList.length; g++){
               success = false;
               //get the current node
               curDom = network.nodes[netOperator.returnNodeIndexById(dominatorList[g], network)];
               solution["rule2"].createStep();
               solution["rule2"].steps[g].text = "Checking node : "+dominatorList[g]+".<br/>";
               //get his dominator neighbors only with higher ids
               domNeighbors = curDom.neighbors.filter(function(elem) {
                    return (network.nodes[netOperator.returnNodeIndexById(elem, network)].dominator)&&(elem>curDom.id);
               });
               //for each one of these neighbors in the previous list
               for(var n=0; n<domNeighbors.length; n++){
                    /*get all the other neighbors of that list and check if the union of their neighbor sets can
                    contain all the neighbors of curDom */
                    for(var t=n+1; t<domNeighbors.length; t++){
                         unionSet = _.union(network.nodes[netOperator.returnNodeIndexById(domNeighbors[n],
network)].neighbors,
                              network.nodes[netOperator.returnNodeIndexById(domNeighbors[t], network)].neighbors );
                         if(_isSubsetOf(curDom.neighbors, unionSet)){
                              solution["rule2"].steps[g].text += "Node's "+dominatorList[g]+" neighborhood is covered by\
                               nodes "+domNeighbors[n]+" and "+domNeighbors[t]+".";
                              success = true;
                              curDom.dominator = false;
                              newDominatorList = newDominatorList.filter(function(index) {
                                   return index != curDom.id;
                              });
                              break;
                         }
                    }
                    if(success){ //curDom's neighborhood is covered. No need for additional checking.
                         break;
                    }
               }
               if(!success){
                    solution["rule2"].steps[g].text += "Node's "+dominatorList[g]+" neighborhood is not covered by two dominator
neighbors.";
               }
               solution["rule2"].steps[g].data = { "dominators" : newDominatorList.slice()};
          }
     }
     else{
          solution["rule2"].createStep();
          solution["rule2"].steps[0].text = "No further computations needed.";
          solution["rule2"].steps[0].data = { "dominators" : newDominatorList.slice()};
     }
     return newDominatorList;
}
```

## Multipoint Relays

Like the Wu and Li algorithm, the purpose of this one is to construct a Minimum Connected Dominating Set using Multipoint Relays. Each network node computes a multipoint relay set with the following properties :

- The multipoint relay set is included in the neighborhood of the node and the elements of the set are called multipoint relays (MPR for short) of that node
- Each 2-hop neighbor of the node has a neighbor in the multipoint relay set (some multipoint relay covers the 2-hop neighbor)

The multipoint relay set plus the node forms a dominating set of the two hop neighborhood of the node.

A multipoint relay selector of a node A is a node that has selected A as a multipoint relay.

### Multipoint Relay Forwarding Rule

A node retransmits the packet only once if it has received the packet the first time from a multipoint relay selector.

### Computing MPR set for a node A with a greedy algorithm:

- Start with an empty MPR set.
- Step 1 : Find the two-hop neighbors that are neighbor only to a single one-hop neighbor of A and add these one-hop neighbors to the MPR set.
- Step 2 : Add in the MPR set the one-hop neighbor of A that covers the largest number of two-hop neighbors that are not covered yet. Repeat this step until all two-hop neighborhood is covered.

### Connected Dominating Set (CDS)

Adding all the MPR nodes to the CDS would create a large set. To reduce the nodes in the set, the algorithm uses the following rules :

A node decides that it is in the Connected Dominating Set if :

- Rule 1 : the node has a smaller ID than all its neighbors
- Rule 2: or it is multipoint relay of its neighbor with the smallest ID.

### Code - The MPR class

It holds the solution objects for each part and the calculate_MPR_CDS method that is called to execute the greedy algorithm.

```
var MPR_cds = function(){
    var that = this;
    that.solution = {
        "MPR_set" : solutionFactory.newSolution(),
        "MPR_cds" : solutionFactory.newSolution()
    };
    that.calculate_MPR_CDS = function(network){
        that.solution["MPR_set"].result["All_MPR_sets"] = _constructMPR(network, that.solution);
        var allMPRs = that.solution["MPR_set"].result["All_MPR_sets"];
        that.solution["MPR_cds"].result["MPR_cds"] = _mprCdsOptimized(that.solution, network, allMPRs);
        return that.solution;
    }
}
```

## Code – Calculation Of MPR Sets

Before explaining how the _constructMPR function is used, let’s see the functions that implement Step 1 and Step 2 from “Computing MPR Set for a Node A”.

The following code executes Step 1. It searches the list of two hop neighbors and for each one it takes the intersection of his neighbor set with the one hop neighbors of the node, whose MPR we are trying to calculate. Function _intersectionOptimized stops the search when it finds two 1-hop neighbors, since we only need to keep the two-hop neighbors covered by 1 one-hop. If the intersection has only one node, we add this one-hop neighbor to the set. Then we calculate the remaining uncovered two-hop neighborhood, by taking the difference of the two-hop neighborhood minus the two-hop neighbors covered by that one-hop. The change boolean variable is used only to detect if no change was made on this step, to write the appropriate text in the solution.

```
/*Used by _calculateTheFirstMprNodes to detect the two hop neighbors that connect with only
one 1-hop. Implements the intersection between the two-hop node's neighbors and the one-hop
neighborhood of the main node we check, to see how many one-hop neighbors cover this two-hop
neighbor. Stops the search when a second 1-hop neighbor is found, since at that point
it is known that the two-hop neighbor is covered by more than 1 one-hop.
*/
 var _intersectionOptimized = function(neighbor_list, one_hop_list){
     let intersect = [];
     for(var i=0; i<neighbor_list.length; i++){
         for(var j=0; j< one_hop_list.length; j++){
             if(neighbor_list[i] == one_hop_list[j]){
                 intersect.push(neighbor_list[i]);
                 if(intersect.length == 2){
                     return intersect;
                 }
             }
         }
     }
     return intersect;
 }

//From all the 2-hop neighbors check with how many of the 1-hop they are connected to.
//If it's only with 1, add this one to the MPR set.
var _calculateTheFirstMprNodes = function(solution, index, network, mpr_set, twoHopNeighbors, oneHopNeighbors){
    var temp2hop = twoHopNeighbors.slice(); //will become the new 2hop neighborhood list
    var change = false;
    var twoHopCovered = [];
    var tempNode;
    var tempNode2;
    var intersect;
    solution["MPR_set"].steps[index].text += "First we will check which of the 2-hop neighbors connect only with one 1-hop
neighbor and then we \
    will add their 1-hop neighbors to the MPR set of the node.</br>";
    for(var g=0; g<twoHopNeighbors.length; g++){
        tempNode = netOperator.returnNodeById(twoHopNeighbors[g], network);
        intersect = _intersectionOptimized(tempNode.neighbors, oneHopNeighbors);
        if((intersect.length == 1) && (_.indexOf(mpr_set, intersect[0]) == -1)){
                change = true;
                mpr_set.push(intersect[0]);
                tempNode2 = netOperator.returnNodeById(intersect[0], network);
                twoHopCovered = _.intersection(tempNode2.neighbors, twoHopNeighbors); //how much 2-hop neighborhood
does that 1-hop cover?
                temp2hop = _.difference(temp2hop, twoHopCovered); //the remaining uncovered 2-hop neighbors
                solution["MPR_set"].steps[index].text += "- Added node "+intersect[0]+" which is the only one covering
"+tempNode.id+".</br>";
        }
    }
    if(!change){
        solution["MPR_set"].steps[index].text += "No change made in this step.</br>";
    }
    return { "twoHop" : temp2hop, "mpr" : mpr_set};
}
```

_calculate2HopStep2 is used to implement Step 2. It keeps a coverage list for each one-hop neighbor that isn’t already in the MPR set. The coverage list refers to how many two-hop neighbors every one-hop neighbor covers. Then by finding the maximum coverage it adds the appropriate node to the set and updates the remaining two-hop neighborhood to be covered.

```
//This is used to add each time in the MPR set the 1-hop neighbors that cover the most 2-hop neighbors
var _calculate2HopStep2 = function(step_index, network, solution, twoHopNeighbors, oneHopNeighbors, mpr_set,
allMPRs){
    var coverageList;
    var tempNode;
    var intersect;
    //Add the 1-hop node with the biggest coverage until all 2-hop neighborhood is covered
    while(twoHopNeighbors.length > 0){
        //Get how many 2-hop nodes are covered by each 1-hop
        //Each entry in the coverageList corresponds to the same indexed entry in oneHopNeighbors
        oneHopNeighbors = _.difference(oneHopNeighbors, mpr_set);
        coverageList = [];
        for(var t=0; t<oneHopNeighbors.length; t++){
            tempNode = netOperator.returnNodeById(oneHopNeighbors[t], network);
            intersect = _.intersection(tempNode.neighbors, twoHopNeighbors);
            coverageList.push({ "nodes" : intersect});
        }
        //Add the node with the maximum coverage to the MPR set
        var max = 0;
        for(var e=0; e<coverageList.length; e++){
            if(coverageList[e]["nodes"].length > coverageList[max]["nodes"].length){
                max = e;
            }
        }
        if(coverageList[max]["nodes"].length > 0){
            mpr_set.push(oneHopNeighbors[max]);
            solution["MPR_set"].steps[step_index].text += "- Added node "+oneHopNeighbors[max]+" with coverage
"+coverageList[max]["nodes"].length+" .<br/>";
            //remove from the 2-hop neighborhood all those who were covered by this node
            twoHopNeighbors = _.difference(twoHopNeighbors, coverageList[max]["nodes"]);
        }
        else{ break;}
    }
    //add this mpr_set to all mprs data
    allMPRs.push(mpr_set);
    solution["MPR_set"].steps[step_index].data = {"mpr_set" : mpr_set};
}
```

The following function uses the previous two, to calculate the MPR set of each node in the network graph.

```
//The main function called to construct the MPR set of each node
var _constructMPR = function(network, solution){
    var twoHopNeighbors;
    var oneHopNeighbors;
    var mpr_set;
    var allMPRs = [];
    var step1Result;
    solution["MPR_set"].text = "Constructing a Connected Dominating Set with Multipoint Relays (MPR).</br> In this phase
we construct the MPR set of each node.";
    //for each node in the network
    for(var i=0; i<network.nodes.length; i++){
        solution["MPR_set"].createStep();
        solution["MPR_set"].steps[i].text = "Calculating MPR set of node : "+network.nodes[i].id+".</br>";
        mpr_set = [];
        oneHopNeighbors = network.nodes[i].neighbors.slice();
        twoHopNeighbors = _return2HopNeighbors(i, network);
        solution["MPR_set"].steps[i].text += "2-Hop Neighbors : "+ _strigifyIntList(twoHopNeighbors)+".</br>";
        if(twoHopNeighbors.length == 0){
            solution["MPR_set"].steps[i].text += "There isn't a 2-hop neighborhood to cover.</br>";
            solution["MPR_set"].steps[i].data = {"mpr_set" : []};
            allMPRs.push([]);
            continue;
        }
        //Add the first 1-hop neighbors that are essential to the set (look at _calculateTheFirstMprNodes() description)
        step1Result = _calculateTheFirstMprNodes(solution, i, network, mpr_set, twoHopNeighbors, oneHopNeighbors);
        twoHopNeighbors = step1Result["twoHop"];
        mpr_set = step1Result["mpr"];
        if(twoHopNeighbors.length > 0){
            solution["MPR_set"].steps[i].text += "</br>2-hop neighbors remaining to cover:
"+_strigifyIntList(twoHopNeighbors)+".</br>";
            solution["MPR_set"].steps[i].text += "Now, each time we will add to the MPR set the 1-hop neighbor who
connects with the most 2-hop neighbors,\
             until all 2-hop neighborhood is covered.</br>";
        }
        else{
            solution["MPR_set"].steps[i].text += "All 2-hop neighborhood covered.</br>";
        }
        //Now add the rest of the 1-hops to cover the entire 2-hop neighborhood (see _calculate2HopStep2() description)
        _calculate2HopStep2(i, network, solution, twoHopNeighbors, oneHopNeighbors, mpr_set, allMPRs);
    }
    return allMPRs;
}
```

The next part is using the rules in “Connected Dominating Set” to create the final CDS.

```
var _mprCdsOptimized = function(solution, network, allMPRs){
    var mprCds = [];
    var smallest;
    solution["MPR_cds"].text = "In this part of the execution, we will define the Connected Dominating Set constructed by
the\
     Multipoint Relays of each node. We use 2 rules to add nodes to the CDS. The node is in the CDS if :</br>\
      Rule 1: the node has the smallest ID of its neighbors</br>\
       Rule 2: or it is multipoint relay of its neighbor with the smallest ID</br>";
    for(var i=0; i<network.nodes.length; i++){
        solution["MPR_cds"].createStep();
        solution["MPR_cds"].steps[i].text = "Checking node "+network.nodes[i].id+" .</br>";
        smallest = true;
        //skip checking if the node is unconnected
        if(network.nodes[i].neighbors.length == 0){
            continue;
        }
        //Rule 1
        for(var j=0; j<network.nodes[i].neighbors.length; j++){
            if(network.nodes[i].id > network.nodes[i].neighbors[j]){
                smallest = false;
                break;
            }
        }
        if(smallest){
            mprCds.push(network.nodes[i].id);
            solution["MPR_cds"].steps[i].text += "Node "+network.nodes[i].id+" has the smallest ID of its neighborhood.\
            We add him to the CDS.</br>";
            solution["MPR_cds"].steps[i].data["dominators"] = mprCds.slice();
        }
        else{
            //Rule 2
            //get the neighbor with the smallest id
            var minId = network.nodes[i].neighbors[0];
            for(var k=1; k<network.nodes[i].neighbors.length; k++){
                if(network.nodes[i].neighbors[k] < minId){
                    minId = network.nodes[i].neighbors[k];
                }
            }
            //see if the current node is a multipoint relay for the node with id = minId
            var nodeIndex = netOperator.returnNodeIndexById(minId, network);
            if(_.indexOf(allMPRs[nodeIndex], network.nodes[i].id) != -1 ){
                mprCds.push(network.nodes[i].id);
                solution["MPR_cds"].steps[i].text += "Node "+network.nodes[i].id+" is a multipoint relay of node " +
                minId+". We add him to the CDS.</br>";
                solution["MPR_cds"].steps[i].data["dominators"] = mprCds.slice();
            }
            else{
                solution["MPR_cds"].steps[i].text += "The node won't be added to the CDS.</br>";
                solution["MPR_cds"].steps[i].data["dominators"] = mprCds.slice();
            }
        }
    }
    return mprCds;
}
```

## DCA (Distributed Clustering Algorithm) Clusters

This algorithm’s aim is to partition the nodes of the network into clusters. It belongs to the clustering solution category, which is described in Chapter 2 –Solution Categories. For each network node, a unique weight is assigned. The nodes with the biggest weights are the most preferable to become clusterheads. All decisions are made by broadcasting messages of two types : CH , declaring that the sender is a clusterhead and JOIN(X,Y), declaring that the sender X wishes to join clusterhead Y. Every time a message is broadcasted, the receivers must keep the information and type of the message and make decisions based on all the messages received so far. Every node has a clusterhead variable set to the id of the node they have joined or themselves, if they are the head of the cluster. The clusterheads must also maintain a list of all the nodes that have joined their cluster. Those that have finished their operations can then stop the communications by executing an EXIT command.

Initially, the nodes with the biggest weight in their neighborhood broadcast CH.

According to the paper used to implement this algorithm, on receiving a CH message the receiver checks if all his neighbors with bigger weights than the sender have sent a JOIN message. If that’s true then he marks the sender as its clusterhead, sends a JOIN message to it and exits. This, though, doesn’t cover some cases when a node has received more than one CH messages and the last message it receives is a CH from a sender with smaller weight than the previous senders of CH messages. To fix this issue, I included an “else” statement after the previous condition check, that checks if all neighbors with bigger weights than the receiver have sent a message of either type. If that is true, the receiver selects as a clusterhead, the one from the senders of CH messages with the biggest weight, sends JOIN and exits.

On receiving a JOIN message, a node checks if it is a clusterhead or not.

If it is, then if the JOIN message was sent with him as the recipient, it adds the sender to its cluster and if all other neighbors with smaller weights than the receiver have sent JOIN messages, it exits.

If it is not a clusterhead, then it checks if all neighbors with bigger weights than it, have sent messages of either type. In that case if they all sent JOIN it becomes a clusterhead and broadcasts CH and immediately after that, it checks if all neighbors with smaller weights than it, have sent JOIN, which means it can exit the operation. If not all neighbors with bigger weights sent JOIN, then it chooses as a clusterhead, the one from the senders of CH messages with the biggest weight, sends JOIN and exits.

The simulation of the execution provided by the code uses the term ‘time-steps”, meaning a group of operations that take place in the same time unit. For each time unit, the messages sent are mentioned at the beginning and the receiving of the messages by the nodes and their decisions at the end. Any new messages to be sent after these decisions are made will be sent at the next time unit, so they will belong to the next time-step. For example time-step one will contain all the CH messages sent at the Initial Process of the Algorithm, along with the decisions of the nodes that received those messages. Time-step 2 will contain all the JOIN and CH messages sent by the nodes that decided to send a message in time-step 1, and so on. In the code, these time-steps are actually instances of the Solution Class from the *Recording Steps Of Execution* section, which record the operations at that time unit, in the steps array of the instances.

The following class defines a solution object (not an instance of the Solution Class in this case) and the calculate_DCA_Clusters function, which given the network graph and the weights of the nodes assigned by the client (either random weights or inserted by the user), performs a series of operations to achieve the final result.

```
//Main object to be returned
var DCA_clusters = function(){
    var that = this;
    that.solution = {
        "final_result" : [],
        "DCA_timesteps" : []
    };
    that.calculate_DCA_Clusters = function(network, extras){
        _changeToDcaNodes(network, extras["weights"]);
        _procedureInit(network, that.solution);
        _stepSimulator(network, that.solution);
        that.solution["final_result"] = _returnClusters(network);
        return that.solution;
    };
}
```

_changeToDcaNodes assigns some new properties to all the nodes, which are necessary to proceed with the rest of the operations. The clusterhead contains, as obvious, who is the clusterhead selected. The cluster array is used only by the clusterheads, which insert into it the nodes that have joined their cluster, plus themselves. CH_messagesInbox and JOIN_messagesInbox are lists containing the CH and JOIN messages received respectively. Each message has a key “sender” assigned to the sender’s id and a key “weight” assigned to the sender’s weight. The toSend property will contain the message to be sent in the next time-step.

```
//Inserts more data to the network object
var _changeToDcaNodes = function(network, weights){
    for(var i=0; i<network.nodes.length; i++){
        network.nodes[i].weight = weights[i];
        network.nodes[i].clusterhead = 0;
        network.nodes[i].cluster = [];
        network.nodes[i].CH_messagesInbox = [];     //will contain the CH messages received.
        network.nodes[i].JOIN_messagesInbox = []; //will contain the JOIN messages received.
        network.nodes[i].toSend = {};   //message to be sent in the next step of the execution.
        network.nodes[i].EXIT = false;
        network.nodes[i].bCastAndExit = false; //when true the node will broadcast first and then exit
        network.nodes[i].iSentCh = false; //it is needed to decide whether a clusterhead has sent a CH already or now is the
time
    }
}
```

The following functions are called when a node sends a message, according to the type. They use an instance of the Solution class, named timestepSolution, which represents the current time-step.

```
//Called when sending a CH message to the neighborhood
var _sendCH = function(node, network, timestepSolution){
    var stepIndex;
    timestepSolution.createStep();
    stepIndex = timestepSolution.steps.length - 1;
    timestepSolution.steps[stepIndex].text = "Node "+node.id+" broadcasted CH.";
    node.iSentCh = true;
    if(node.clusterhead != node.id){
        node.clusterhead = node.id;
        node.cluster.push(node.id);
    }
    timestepSolution.steps[stepIndex].data["clusters"] = _returnClusters(network);
}

//Called when sending a JOIN message to the neighborhood
var _sendJOIN = function(node, clusterhead, network, timestepSolution){
    var stepIndex;
    timestepSolution.createStep();
    stepIndex = timestepSolution.steps.length - 1;
    timestepSolution.steps[stepIndex].text = "Node "+node.id+" broadcasted JOIN ("+node.id+","+clusterhead.id+") and
Exited.";
    node.EXIT = true;
    timestepSolution.steps[stepIndex].data["clusters"] = _returnClusters(network);
}
```

This is the implementation of the On Receive CH function.

```
//Called when a node receives CH
var _receiveCH = function(node, sender, network, timestep, timestepSolution){
    var biggerWeightNeighbors = [];
    var tempNode;
    var stepIndex;
    timestepSolution.createStep();
    stepIndex = timestepSolution.steps.length - 1;
    timestepSolution.steps[stepIndex].text = "Node "+node.id+" received CH from "+sender.id+".";
    node.CH_messagesInbox.push({"sender" : sender.id, "weight" : sender.weight});
    //get all the neighbors with bigger weights than the sender
    for(var i=0; i<node.neighbors.length; i++){
        tempNode = netOperator.returnNodeById(node.neighbors[i], network);
        if(tempNode.weight > sender.weight){
            biggerWeightNeighbors.push(tempNode);
        }
    }
    //if they all have sent JOIN messages then join this sender's cluster
    var allSent = true;
    for(var k=0; k<biggerWeightNeighbors.length; k++){
        if(!_checkIfNodeSent("JOIN", node, biggerWeightNeighbors[k])){
            allSent = false;
            break;
        }
    }
    if(allSent){
        node.clusterhead = sender.id;
        //send JOIN to sender
        node.toSend = {"type" : "JOIN", "receiver" : sender.id, "timestep" : (timestep+1)};
    }
    else{
        //get the neighbors with bigger weight than mine and check if they have all sent CH or JOIN
        let conditionOR = false;
        for(var i=0; i<node.neighbors.length; i++){
            tempNode = netOperator.returnNodeById(node.neighbors[i], network);
            if(tempNode.weight > node.weight){
                conditionOR = _checkIfNodeSent("CH", node, tempNode) || _checkIfNodeSent("JOIN", node, tempNode);
                if(!conditionOR){
                    break;
                }
            }
        }
        if(conditionOR){
            //they all sent CH or JOIN
            //choose the clusterhead with the biggest weight and JOIN him
            node.clusterhead = _getBiggestClusterhead(node.CH_messagesInbox);
            node.toSend = {"type" : "JOIN", "receiver" : node.clusterhead, "timestep" : (timestep+1)};
            timestepSolution.steps[stepIndex].text += "Node "+node.id+" sees that all the neighbors with bigger weights have"
+
            " sent either CH or JOIN. In the next step he will send JOIN to the clusterhead neighbor with the biggest weight, 
which is "
            +node.clusterhead+
            " .";
        }
    }
    timestepSolution.steps[stepIndex].data["clusters"] = _returnClusters(network);
}
```

_clusterheadHandlesJOIN is used to separate the code from _receiveJOIN and make her smaller in size. It handles the case of a clusterhead receiving a JOIN message. In _receiveJOIN conditionOR is true if all nodes with bigger weight than the receiver have sent either CH or JOIN and conditionJOIN is true when all of them sent JOIN.

```
//How a clusterhead handles a JOIN message
var _clusterheadHandlesJOIN = function(receiver, sender, clusterhead, smallerWeightNeighbors, network, step){
    //if  the JOIN is for me
    if(clusterhead.id == receiver.id){
        receiver.cluster.push(sender.id);
        step.text += "Clusterhead "+receiver.id+" put node "+sender.id+" to its cluster.</br>";
    }
    //if all the smaller weighted neighbors joined someone else exit
    var allSent = true;
    for(var k=0; k<smallerWeightNeighbors.length; k++){
        if(!_checkIfNodeSent("JOIN", receiver, smallerWeightNeighbors[k])){
            allSent = false;
            break;
        }
    }
    if(allSent){
        if(receiver.iSentCh){
            receiver.EXIT = true;
            step.text +="Clusterhead "+receiver.id+" Exited. All smaller weighted neighbors have sent JOIN.";
        }
        else{
            receiver.bCastAndExit = true;
            step.text +="Clusterhead "+receiver.id+" will send CH and Exit. All smaller weighted neighbors have sent JOIN.";
        }
    }
}
```

```
//Called when a node receives JOIN
var _receiveJOIN = function(receiver, sender, clusterhead, network, timestep, timestepSolution){
    var smallerWeightNeighbors = [];
    var tempNode;
    var conditionOR; //boolean
    var conditionJOIN; //boolean
    var stepIndex;
    timestepSolution.createStep();
    stepIndex = timestepSolution.steps.length - 1;
    timestepSolution.steps[stepIndex].text = "Node "+receiver.id+" received JOIN ("+sender.id+","+clusterhead.id+").</br>";
    receiver.JOIN_messagesInbox.push({"sender": sender.id, "clusterhead" : clusterhead.id});
    //get the neighbors with smaller weight than me, we are going to need them later
    for(var i=0; i<receiver.neighbors.length; i++){
        tempNode = netOperator.returnNodeById(receiver.neighbors[i], network);
        if(tempNode.weight < receiver.weight){
            smallerWeightNeighbors.push(tempNode);
        }
    }
    //if i am a clusterhead
    if(receiver.clusterhead == receiver.id){
        _clusterheadHandlesJOIN(receiver, sender, clusterhead, smallerWeightNeighbors, network,
timestepSolution.steps[stepIndex]);
    }
    else{
        //get the neighbors with bigger weight than mine and check if they have all sent CH or JOIN
        conditionOR = false;
        conditionJOIN = true;
        for(var i=0; i<receiver.neighbors.length; i++){
            tempNode = netOperator.returnNodeById(receiver.neighbors[i], network);
            if(tempNode.weight > receiver.weight){
                conditionOR = _checkIfNodeSent("CH", receiver, tempNode) || _checkIfNodeSent("JOIN", receiver,
tempNode);
                conditionJOIN = conditionJOIN && _checkIfNodeSent("JOIN", receiver, tempNode);
                if(!conditionOR){
                    break;
                }
            }
        }
        if(conditionOR){
            if(conditionJOIN){ //if they all sent JOIN
                //become clusterhead
                receiver.cluster.push(receiver.id);
                receiver.clusterhead = receiver.id;
                receiver.toSend = {"type" : "CH", "timestep" : (timestep+1)};
                timestepSolution.steps[stepIndex].text += "Node "+receiver.id+" became clusterhead. All bigger weighted
neighbors have sent JOIN.";
                //if those with smaller weight have sent JOIN exit
                var exit = true;
                for(var n=0; n<smallerWeightNeighbors.length; n++){
                    if(!_checkIfNodeSent("JOIN", receiver, smallerWeightNeighbors[n])){
                        exit = false;
                        break;
                    }
                }
                if(exit){
                    receiver.bCastAndExit = true;
                    timestepSolution.steps[stepIndex].text += "</br>Node "+receiver.id+" will send CH and Exit.";
                }
            }
            else{ //they all sent CH or JOIN
                //choose the clusterhead with the biggest weight and JOIN him
                receiver.clusterhead = _getBiggestClusterhead(receiver.CH_messagesInbox);
                receiver.toSend = {"type" : "JOIN", "receiver" : receiver.clusterhead, "timestep" : (timestep+1)};
                timestepSolution.steps[stepIndex].text += "Node "+receiver.id+" sees that all the neighbors with bigger weights
have" +
                " sent either CH or JOIN. In the next step he will send JOIN to the clusterhead neighbor with the biggest
weight, which is "
                +receiver.clusterhead+
                " .";
            }
        }
    }
    timestepSolution.steps[stepIndex].data["clusters"] = _returnClusters(network);
}
```

In _procedureInit, the initial process of the algorithm is executed. All the nodes with the biggest weight in their neighborhood will broadcast CH.

```
//Initial procedure : Those with the biggest ids in their neighborhood send CH
var _procedureInit = function(network, solution){
    var greaterWeight;
    var receiveQueue = [];
    var neighbors = [];
    var timestep1 = solutionFactory.newSolution();
    solution["DCA_timesteps"].push(timestep1);
    timestep1.text ="Timestep 1.</br>Beginning with \"Procedure Init\" of the DCA Algorithm.\
    </br>Only the nodes with the highest weight in their neighborhood send CH.";
    //First send the messages
    for(var i=0; i<network.nodes.length; i++){
        neighbors = netOperator.returnNeighborObjects(network.nodes[i], network);
        greaterWeight = true;
        for(var k=0; k<neighbors.length; k++){
            if(network.nodes[i].weight < neighbors[k].weight){
                greaterWeight = false;
                break;
            }
        }
        if(greaterWeight){
            _sendCH(network.nodes[i], network, timestep1);
            for(var k=0; k<neighbors.length; k++){
                receiveQueue.push({"nodeToCall" : neighbors[k], "sender" : network.nodes[i]});
            }
        }
    }
    //Now call the receive methods
    for(var i=0; i<receiveQueue.length; i++){
        _receiveCH(receiveQueue[i]["nodeToCall"], receiveQueue[i]["sender"], network, 1, timestep1);
    }
}
```

Finally, _stepSimulator is called to simulate the execution in time units, as described in above all the code snippets. At the beginning of a time unit, all pending messages are sent and at the end, the nodes start receiving the broadcasted messages. This works by using the receiveQueue list, to hold the information needed to call the receiving functions of all the nodes that “hear” the broadcasted messages, at the end of the time unit.

```
//It simulates the steps of execution for the algorithm after _procedureInit
var _stepSimulator = function(network, solution){
    var node;
    var neighbors;
    var timestep = 2;
    var timestepSol; //will contain the solution of the current timestep
    var receiveQueue = [];  //In an execution step receiving a message happens at the end, after sending all messages pending
so far.
                                        //So we use a queue to track which receive functions to call after all nodes have sent their messages.
    var wereMessagesSent=true; //were any messages sent in this step? If not stop to avod infinite looping when errors in the
algorithm occur
    while((!_haveAllExited(network))&&wereMessagesSent){
        wereMessagesSent = false;
        timestepSol = solutionFactory.newSolution();
        solution["DCA_timesteps"].push(timestepSol);
        timestepSol.text ="Timestep "+timestep+".</br>";
        receiveQueue = [];
        //First send all pending messages for this step
        for(var i=0; i<network.nodes.length; i++){
            node = network.nodes[i];
            if(!node.EXIT){
                if(node.toSend["type"] != null){
                    neighbors = netOperator.returnNeighborObjects(node, network);
                    if((node.toSend["type"] == "CH") && (node.toSend["timestep"] == timestep)){
                        wereMessagesSent = true;
                        _sendCH(node, network, timestepSol);
                        for(var j=0; j<neighbors.length; j++){
                            if(!neighbors[j].EXIT){
                                receiveQueue.push({"type" : "CH", "nodeToCall" : neighbors[j], "sender" : node});
                            }
                        }
                        node.toSend = {};
                    }
                    else if((node.toSend["type"] == "JOIN") && (node.toSend["timestep"] == timestep)){
                        wereMessagesSent = true;
                        _sendJOIN(node, netOperator.returnNodeById(node.toSend["receiver"], network), network,
timestepSol);
                        for(var j=0; j<neighbors.length; j++){
                            if(!neighbors[j].EXIT){
                                receiveQueue.push({"type" : "JOIN", "nodeToCall" : neighbors[j], "sender" : node,
                                "receiver" : netOperator.returnNodeById(node.toSend["receiver"], network)});
                            }
                        }
                        node.toSend = {};
                    }
                    if(node.bCastAndExit){
                        node.EXIT = true;
                        timestepSol.steps[timestepSol.steps.length-1].text += " Node "+node.id+" Exited.";
                    }
                }
            }
        }
        //Now execute the receiving functions
        for(var i=0; i<receiveQueue.length; i++){
            if((receiveQueue[i]["type"] == "CH")&&(!receiveQueue[i]["nodeToCall"].EXIT)){
                _receiveCH(receiveQueue[i]["nodeToCall"], receiveQueue[i]["sender"], network, timestep, timestepSol);
            }
            else if((receiveQueue[i]["type"] == "JOIN")&&(!receiveQueue[i]["nodeToCall"].EXIT)){
                _receiveJOIN(receiveQueue[i]["nodeToCall"], receiveQueue[i]["sender"], receiveQueue[i]["receiver"],
network, timestep, timestepSol);
            }
        }
        timestep ++;
    }
}
```

## Max-Min D-Cluster Formation

This one, like DCA, tries to partition the network graph into clusters too. It uses a number **d**, which denotes that the clusterheads selected at the end of the execution will form a d-hop dominating set and the ordinary nodes will be at most d-hops away from a clusterhead. The whole process described by the authors, runs for 2d rounds of information exchange. For every round, each node maintains the winner node’s id for that round and the id of the sender node, that sent the message containing the winner id. The first d rounds are called the **floodmax** and the next d rounds are the **floodmin**. In the code, each node object has two lists as properties, one for the floodmax winners and respective senders and one for the floodmin ones.

Initially all nodes consider their own id as the winner.

### Floodmax

In these **d** rounds of execution each node broadcasts its winner value to its 1-hop neighbors. After it has heard from all neighbors in a single round, it chooses as the new winner, the largest between his own winner value and the ones sent by the neighbors. In the next round it broadcasts the new winner value and so on.

### Floodmin

The process is the same as in floodmax, but now the winner id is the smallest between the node’s own value and the ones sent by the neighbors.

### Clusterhead Selection

After the 2d rounds of messaging are finished, the following rules are applied with this sequence to each node, to determine who will be its clusterhead.

- Rule 1 : first the node checks if it has received its own id in the 2nd d rounds. If so, it declares itself as a clusterhead.
- Rule 2 : if rule 1’s condition failed, it checks for node pairs. Node pair is any node id that occurs at least once as a winner in both floodmax and floodmin for an individual node. The node selects its minimum node pair to be the clusterhead. If no such pairs exist, proceed to Rule 3.
- Rule 3 : select the maximum id of the floodmax as the clusterhead.

After this procedure the authors of *"Max-Min D-Cluster Formation in Wireless Ad Hoc Networks"* (see **References**) propose that to form the final clusters and inform the clusterheads of who joined them, a convergecast starts from the gateway nodes and the fringes of the cluster, using the sender values to backtrack the origins of the message back to the clusterhead. During this backtracking, each node in the path adds its own id, so at the end, the clusterhead has heard from all neighbors. This also makes sure that in scenarios where there is another clusterhead in the path during backtracking, that clusterhead adopts and informs the sending node. It has been observed by the authors of *"On multihop clusters in wireless sensor networks"* that the convergecast solution causes infinite loops in some cases. To avoid such issues, after applying the above rules, we use the solution proposed by the latter paper instead of the convergecast, which denotes that the clusterheads start sending messages to their neighborhood to join their cluster and the receivers rebroadcast the messages, as long as that doesn’t exceed the d-hops from the cluster. A node accepts as its clusterhead, the one whose message arrived first to it. This solution doesn’t need Rule 2 and 3 to be applied, but in this application we show both what the nodes have decided by using all the above rules, plus what they would have decided if only the latter messaging solution was used. There are two different buttons that color the clusters formed according to the solution selected.

The results of floodmax and floodmin are presented in a table, each column of which corresponds to the node with the id written in the first row. Below the table are the steps of the messaging solution. Since simulating real conditions of wireless communication is not the purpose of this app, the order of the messages broadcasted is dependent on the ordering of the nodes inside the nodes list of the network object, as received by the client.

The class used for storing data and calling the appropriate methods. The calculateMaxMixClusters takes the network object and the number d as inputs and returns the solution object with its data filled with the execution results. The array clusters will keep the result after the application of the above rules, while the clusters2 array will keep the result after the messaging solution is applied.

```
//Main Object To Be Returned
var MaxMinClusters = function(){
   var that = this;
   that.solution = {
      "floodmax" : [],
      "floodmin" : [],
      "clusters" : [], //will have the clusters formed after floodmax/floodmin
      "clusters2" : [], //will have the clusters formed after the messaging process
      "messages_solution" : solutionFactory.newSolution(),
      "extra_notes" : "",

   };
   that.calculateMaxMixClusters = function(network, d){
      //the d value is assosiated with the hops in the algorithm
      _convertToMaxMinNodes(network);
      _roundSimulator(network , d);
      _finalWinners(network, d, that.solution);
      _constructSolution(network, that.solution, d);
      _clustersAfterMessaging(network, that.solution, d);
      return that.solution;
   };
}
```

_convertToMaxMinNodes simply adds the floodmax and floodmin tables, plus the sender and the final winner variables, all as new properties to each node object of the network. _procedureInit below, is called inside _roundSimulator at the first round of the floodmax, where each node broadcasts each own id as the winner id.

```
//Initial procedure
var _procedureInit = function(network){
   for(var i=0; i<network.nodes.length; i++){
      network.nodes[i].finalWinner = {"sender" : network.nodes[i].id, "winner" : network.nodes[i].id};
      _broadcastWinner(network.nodes[i], network.nodes[i].id, network);
   }
   for(var i=0; i<network.nodes.length; i++){
      _evaluateWinner(network.nodes[i], "max");
   }
}
```

The basic implementation of the algorithm takes place in the function below. The comments show were each part of the execution takes place. After this process, the final winners are decided for each node by calling _finalWinners. Inside the latter function, _returnFloodIntersection returns the node pairs for this specific node by taking an intersection of the floodmax and floodmin arrays, using the winners for comparison. Also, _returnListWinner will return either the max or the minimum of a winners list.

```
//Round simulator
var _roundSimulator = function(network , d){
    //floodmax
    for(var j=0; j<d; j++){
        if(j == 0){
            _procedureInit(network);
        }
        else{
            for(var i=0; i<network.nodes.length; i++){
                //broadcast previous round winner
                _broadcastWinner(network.nodes[i], network.nodes[i].floodmax[j-1]["winner"], network);
            }
            for(var i=0; i<network.nodes.length; i++){
                _evaluateWinner(network.nodes[i], "max");
            }
        }
    }
    //floodmin
    for(var j=0; j<d; j++){
        if(j == 0){
            for(var i=0; i<network.nodes.length; i++){
                //broadcast previous round winner
                _broadcastWinner(network.nodes[i], network.nodes[i].floodmax[d-1]["winner"], network);
            }
            for(var i=0; i<network.nodes.length; i++){
                _evaluateWinner(network.nodes[i], "min");
            }
        }
        else{
            for(var i=0; i<network.nodes.length; i++){
                _broadcastWinner(network.nodes[i], network.nodes[i].floodmin[j-1]["winner"], network);
            }
            for(var i=0; i<network.nodes.length; i++){
                _evaluateWinner(network.nodes[i], "min");
            }
        }
    }
}
```

```
//Will decide the final winners of the execution
var _finalWinners = function(network, d, solution){
    //Use rule 1,2 and 3 to define th winners
    for(var i=0; i<network.nodes.length; i++){
        //Rule 1 : I received my id in floodmin, so i become clusterhead
        let index = _isNodeInFloodList(network.nodes[i].id, network.nodes[i].floodmin);
        if( index != -1){
            network.nodes[i].finalWinner = network.nodes[i].floodmin[index];
        }
        else{//Rule 2 select minimum node pair if it exists
            var nodePairs = _returnFloodIntersection(network.nodes[i]);
            if(nodePairs.length > 0){
                network.nodes[i].finalWinner = _returnListWinner("min", nodePairs);
            }
            else{ //No node pairs exist, select maximum id in floodmax
                network.nodes[i].finalWinner = _returnListWinner("max", network.nodes[i].floodmax);
            }
        }
    }
}
```

For simulating the messages sent by the clusterheads at the end of the process, we use _clustersAfterMessaging. Every node object gets a new message property, to hold only the first message received. To prevent transmitting a message beyond d-hops from the clusterhead that sent it, we use a “hop” value inside the message objects. Each time a message is rebroadcasted, its hop value is increased by one before sending, so the receivers know if they should retransmit or not, by checking this value.

```
/*
After the floodmax and floodmin stages, the clusterheads broadcast a message to notify the other nodes to join their cluster.
These messages are rebroadcasted by the receiving nodes, for a maximum of d-hops away from the clusterhead. The receiving
nodes also
choose as clusterhead, the one whose message reached them first. This process replaces the
convergecast solution originally proposed, because the latter leads to infinite loops in some occassions.
The following function implements the process.
*/
var _clustersAfterMessaging = function(network, solution, d){
   let toSend = []; //a list with the nodes to send and what to send
   var clusters = solution.clusters;
   var messageSol = solution.messages_solution;
   for(var i=0; i<network.nodes.length; i++){
      network.nodes[i].message = { sender : -1, hop : -1, clusterhead: -1}; //keeps the first message each node receives. Every
other message is dropped
   }
   for(var i=0; i<clusters.length; i++){
      let node = netOperator.returnNodeById(clusters[i].clusterhead, network);
      node.message = { sender : 0, hop : 0, clusterhead : node.id};
      toSend.push(node.id); //the clusterheads will begin broadcasting
      solution.clusters2.push({clusterhead : node.id, group:[node.id]});
   }
   while(toSend.length > 0){
      let toSendNew = [];
      for(var i=0; i<toSend.length; i++){
         let node = netOperator.returnNodeById(toSend[i], network);
         if(node.message.hop < d){ //broadcast only if the message doesn't reach beyond the d-hops
            messageSol.createStep();
            messageSol.steps[messageSol.steps.length - 1].text = "Node "+node.id+" broadcasts a 'Join Clusterhead's
"+node.message.clusterhead+" Cluster' message.";
            messageSol.steps[messageSol.steps.length - 1].data = _deepCopyClusters(solution.clusters2);
            for( var j=0; j<node.neighbors.length; j++){ //broadcast to the neighbors
               let neighID = node.neighbors[j];
               let neighbor = netOperator.returnNodeById(neighID, network);
               if(neighbor.message.sender == -1){ //nobody sent me a message so far
                  neighbor.message = {sender : node.id, hop : node.message.hop+1, clusterhead : node.message.clusterhead};
                  toSendNew.push(neighID);    //he has to broadcast on the next round
                  _joinClusterhead(neighID, node.message.clusterhead, solution);   //and join the cluster by the way
                  messageSol.createStep();
                  messageSol.steps[messageSol.steps.length - 1].text = "Node "+neighID+" accepts and joins Clusterhead
"+node.message.clusterhead+".";
                  messageSol.steps[messageSol.steps.length - 1].data = _deepCopyClusters(solution.clusters2);
               }
            }
         }
      }
      toSend = toSendNew;
   }
}
```

# Maximal Independent Set (MIS)

It results in the construction of a Connected Dominating Set by using a rooted spanning tree. Given a graph with V vertices and E edges, an independent set is a subset of the vertices, such as no two nodes in this set are adjacent. The independent set is maximal if no node can be added without violating this independence. In this MIS, the distance between any pair of its complementary subsets is guaranteed to be exactly two hops. The CDS is constructed after marking all nodes either black or grey, based on message exchanges, using the spanning tree to create a hierarchy and assign to each node a unique rank. The whole process is divided in three parts :

1) Level Marking
2) Color Marking
3) Construction Of CDS

## Rooted Spanning Tree

In this application, the root of the spanning tree is selected by the user instead of using a leader election algorithm. For the construction of the spanning tree though, the PIF (Propagation Information with Feedback) algorithm proposed in *"Propagation and Leader Election in a Multihop Broadcast Environmen*t" is implemented, starting from the node that the user has selected. PIF starts with the root broadcasting a message and the intermediate nodes recording the sender of the first message they got and rebroadcasting that message until it reaches the leafs of the tree. Then the leafs start sending acknowledgements to their parent and when a node gets acknowledgements from all its children, it sends an acknowledgement to its own parent, until the root has heard from all its neighbors.

In this software the order of the messages sent is dependent on the ordering of the nodes as inserted by the user. The description of the steps (but not the results) of this part of the execution is omitted, because the tree is not considered a main part of the MIS algorithm, but an input instead.

## Level Marking Process

After the construction of the tree, the MIS algorithm induces a ranking from it, which is the total ordering of pairs (level, ID) for each node in the lexicographic order. Then the level marking process begins. Each node must contain the following variables for this part :

- x1, denoting the number of neighbors whose levels are not known yet, initialized to the number of neighbors.
- x2, to count the number of children that have not reported completion yet, initialized to the number of children.
- levelList, that keeps the levels of each neighbor, initially empty
- y, to store the number of lower ranked neighbors

The root of the spanning tree broadcasts a LEVEL message containing the level of the root and each receiving node rebroadcasts it, sending their own level, until it reaches the leaves. At this point every node has calculated their own level and the leaves can start sending a LEVEL_COMPLETE message to their parent.

The calculation of the levels happens as follows :

- On receiving a LEVEL message, a node decreases x1 by 1 and appends the sender's ID and level to the levelList. If x1 is 0 after decreasing, it sets y to the number of lower ranked neighbors, induced from the levelList. It also checks if the message was sent from its parent. If true it calculates its own level by incrementing the sender's level by 1 and broadcasts a LEVEL message with his own level.
- When the leaves have calculated their own level, they broadcast LEVEL_COMPLETE to their parent. On receiving a LEVEL_COMPLETE message, a node decreases x2 by 1. If x2 is 0 after the operation and it's not the root, it transmits a LEVEL_COMPLETE message to its parent. Then it resets x2 to the number of children. The process stops when the root has heard such a message from all its neighbors and resets x2 as well.

## Color Marking Process

All nodes are initially marked with white color and will eventually be marked gray or black. Each node must also maintain a blackList, keeping the IDs of at most five black neighbors. After the Level Marking Process, the root marks itself black and broadcasts a BLACK message.

- Upon receiving a BLACK message, the receiver appends the sender's ID to the blackList and if it is white, it becomes gray and sends GRAY with its level.
- Upon receiving GRAY, a white node checks if the rank of the sender is lower than its own (smaller level or same level and smaller ID). If true then it decrements y by one and if y becomes 0, it marks itself as black and broadcast BLACK.

- When a leaf node becomes either grey or black, it sends a MARK_COMPLETE message to its parent. When such a message is received, the receiver decrements x2 by 1 and if it becomes 0, it sends MARK_COMPLETE to the parent until the root has again hear from all its neighbors.

## Construction Of CDS

After color marking is complete, the dominating tree (different from the spanning tree used above) is constructed. For reference, let’s name the dominating tree T. Every node has a variable z, which denotes if it is on T or not and initially set to “false”, another “parent” property, keeping the ID of the parent inside T and a list called childrenList, keeping the children of the node in T. Two more variables “degree” and “root” are also used only by the root, the first one initialized to 0. The process is the following :

- The root of the spanning tree resets x1 to the number of neighbors and broadcasts a QUERY message.
- Upon receiving a QUERY message, if the node is gray, it replies to the sender with a REPORT message, containing the number of the black neighbors recorded in blackList, during the color marking.
- When the root receives a REPORT message, it decrements x1 by 1 and if the number of black neighbors of the sender is greater than the degree, it sets degree to that number and the root variable to the sender’s id. If x1 has become 0, the root transmits a ROOT message to the sender whose id is stored in the root variable. The receiver knows that it was selected as the root of T, so it sets z to “true” and broadcasts an INVITE2 message.
- Upon receiving an INVITE2 message, a black node with z set to “false”, sets z to “true” and “parent” to the id of the sender. Then it transmits a JOIN to the sender (who will add the node to its childrenList) and then it broadcasts an INVITE1 message.
- When an INVITE1 message has been received, a gray node with z “false”, will set z to “true” and record in “parent” the id of the sender. It will then transmit a JOIN, as described above, and then will broadcast an INVITE2 message.

When the messages stop, to Dominating Tree will have been constructed.

## The Code

This is the class that defines the solution object to be returned and the method to call to construct a MIS on a given graph and with a given root. _convertToMisNodes adds to each node the properties (variables) mentioned in the processes above, that are needed for the calculations.

```
var misObject = function(){
   that = this;
   that.solution = {
      "edges" : [],
      "levels" : solutionFactory.newSolution(),
      "colors" : solutionFactory.newSolution(),
      "cds" : solutionFactory.newSolution(),
      "cdsRootIndex" : -1,
      "cdsRootColoringStep" : -1
   };
   that.constructMIS = function(network, rootNode){
        var index = netOperator.returnNodeIndexById(rootNode, network); //get the index of the root inside the network.nodes
array
      _convertToMisNodes(index, network);    //add the appropriate properties to the network nodes
      that.solution["edges"] = _constructRootedTree(network); //the network object is transformed to a rooted tree
      _beginMessaging(network, that.solution);
      return that.solution;
   };
}
```

This function is used to construct a rooted spanning tree, using the root given by the user. After the root sends to its neighbors, every neighbor is added to the toBroadcastList array. This array is used to keep the nodes that will transmit their messages on the next step. In the while loop, while there are nodes to broadcast, for every such node the neighbors are obtained and if they haven't already marked who their parent is, they mark the sender as the parent and the sender marks them as children. As mentioned above, this step is not part of the main algorithm, so the sending of the messages is not simulated here, only the variables of the nodes change when it is considered that they received a message.

```
//Every node must have parent/children - use of the PIF algorithm (see references in README)
var _constructRootedTree = function(network){
   var edges = []; //Will be used for visually representing the tree
   var toBroadcastList = []; //It will hold which nodes will broadcast next
   //First the root sends to all its neighbors
   var neighbors = netOperator.returnNeighborObjects(network.nodes[network.rootIndex], network);
   for(var i=0; i<neighbors.length; i++){
      neighbors[i].parent = network.nodes[network.rootIndex].id;
      network.nodes[network.rootIndex].children.push(neighbors[i].id);
      network.nodes[network.rootIndex].childrenNotComplete ++;
      toBroadcastList.push(neighbors[i]);
      edges.push({"source" : network.nodes[network.rootIndex].id, "target" : neighbors[i].id});
   }
   //Then the rest of the nodes broadcast
   var newBcastList;
   while(toBroadcastList.length > 0){
      newBcastList = [];
      for(var i=0; i<toBroadcastList.length; i++){
         neighbors = netOperator.returnNeighborObjects(toBroadcastList[i], network);
         for(var j=0; j<neighbors.length; j++){
            if(neighbors[j].parent == -1){  //if this neighbor doesn't have a parent already
               neighbors[j].parent = toBroadcastList[i].id;
               toBroadcastList[i].children.push(neighbors[j].id);
               toBroadcastList[i].childrenNotComplete ++;
               newBcastList.push(neighbors[j]);
               edges.push({"source" : toBroadcastList[i].id, "target" : neighbors[j].id});
            }
         }
      }
      toBroadcastList = newBcastList;
   }
   return edges;
}
```

_beginMessaging is used to run the whole process. It splits the solution to three parts, two for the marking processes and the last one for constructing the CDS. A message queue has been added to the network object to keep track of the messages to be sent in each step. The function _sendMessage (ommited here along with others due to large size) actually uses a switch case to decide what actions to perform according to the type of message to be sent. It calls the appropriate functions to handle receiving messages, which in turn update the network message queue, when a new message should be transmitted.

```
//Run the algorithm by sending messages
var _beginMessaging = function(network, solution){
    //Level Marking process ==========================================
    solution["levels"].text = "<p class=\"solution-heading\"><strong>Part 1 :</strong> First we begin with the Level Marking
process. Given the rooted spanning tree, the root broadcasts a LEVEL message first"+
    " and the process continues until all nodes have calculated their own levels.</p>";
    //root sends LEVEL message to the children first
    var rootNode = network.nodes[network.rootIndex];
    _sendMessage("level", rootNode, null, network, solution["levels"]);
    //then the rest of the nodes broadcast a LEVEL message
    var message;
    var sender;
    while(!network.beginColorMarking){
        message = network.messageQueue.shift(); //remove and return the first element
        sender = netOperator.returnNodeById(message.sender, network);
        switch(message.type){
            case "level":
                _sendMessage("level", sender, null, network, solution["levels"]);
                break;
            case "levelComplete":
                _sendMessage("levelComplete", sender, null, network, solution["levels"]);
                break;
            default : break;
        }
    }
    //Color marking process ==========================================
    solution["colors"].text = "<p class=\"solution-heading\"><strong>Part 2 :</strong> Now we continue with the color
marking process. All nodes are initially marked with white color and will be marked with either"+
    " gray or black eventually.</p>";
    rootNode.color = "black";
    _sendMessage("black", rootNode, null, network, solution["colors"]);
    while(!network.beginCDS){
        message = network.messageQueue.shift(); //remove and return the first element
        sender = netOperator.returnNodeById(message.sender, network);
        switch(message.type){
            case "black":
                _sendMessage("black", sender, null, network, solution["colors"]);
                break;
            case "gray":
                _sendMessage("gray", sender, null, network, solution["colors"]);
                break;
             case "markComplete":
                 _sendMessage("markComplete", sender, null, network, solution["colors"]);
                break;
            default : break;
        }
    }
    //keep the colors of the nodes at their current state as the final result of this process
    solution["colors"].data = _returnColorArray(network);
    //CDS construction ===================================================================
    solution["cds"].text = "<p class=\"solution-heading\"><strong>Part 3 :</strong> Now the construction of a Connected
Dominating Set Tree will begin. The tree's root will be the gray"+
    " neighbor of the spanning tree root with the largest number of black neighbors.</p>";
    solution["cds"].result.edges = [];
    rootNode.nonLeveledNeighbors = rootNode.neighbors.length;
    _sendMessage("query", rootNode, null, network, solution["cds"]);
    var receiver;
    while(network.messageQueue.length > 0){
        message = network.messageQueue.shift(); //remove and return the first element
        sender = netOperator.returnNodeById(message.sender, network);
        switch(message.type){
            case "query":
                _sendMessage("query", sender, null, network, solution["cds"]);
                break;
            case "report":
                receiver = netOperator.returnNodeById(message.receiver, network);
                _sendMessage("report", sender, receiver, network, solution["cds"]);
                break;
            case "root":
                receiver = netOperator.returnNodeById(message.receiver, network);
                _sendMessage("root", sender, receiver,  network, solution["cds"]);
                solution["cdsRootColoringStep"] = solution["cds"].steps.length;
                break;
            case "invite2":
                _sendMessage("invite2", sender, null, network, solution["cds"]);
                break;
            case "invite1":
                _sendMessage("invite1", sender, null, network, solution["cds"]);
                break;
            case "join" :
                receiver = netOperator.returnNodeById(message.receiver, network);
                _sendMessage("join", sender, receiver, network, solution["cds"]);
                break;
            default : break;
        }
    }
    solution["cdsRootIndex"] = network.cdsRootIndex;
    solution["cds"].result.edges = solution["cds"].steps[solution["cds"].steps.length - 1].data.edges.slice();
}
```

## Local Minimum Spanning Tree (LMST)

Provides a topology solution, by creating local MSTs for each node independently. It is composed of the phases of information collection, topology construction and determination of transmission power and an optional phase of optimization, making all edges bidirectional. At this application we don’t deal with transmission powers, so that phase is omitted.

### Information Exchange

Every node must know which nodes it can reach with its maximal transmission power. This process refers to defining the 1-hop neighborhood for each node, which in the app is inserted by the user.

### Topology Construction

Each node applies Prim’s algorithm independently to obtain its local minimum spanning tree. The graph as input for Prim, has as vertices the node and its 1-hop neighbors and as edges all the connections with each other. Prim uses distances to decide which edge to include in each step. Here to make every option unique, if some edges have the same distance, then for all of them we calculate the maximum(source ID, target ID) and use the edge with the minimum such value. If there are still more than one edges with this value, then we calculate the minimum(source ID, target ID) and choose the edge with the minimum such value. The final topology graph is not a superposition of the nodes’ individual LMSTs. An edge starting from a node A and ending to a node B is included in the final graph, only if B is a 1-hop neighbor of A, in A’s LMST. A bi-directional edge is formed when both nodes have edges towards each other in the final graph. Otherwise the edge is uni-directional.

### Optimization Phase

At the end of topology construction, we can choose how to deal with uni-directional edges, if they exist. There are two options, either make these edges bi-directional (in real application the node with such an edge sends a message to the target node to add this edge in its tree too), which is called the G0+ graph, or to delete all uni-directional links, which is called G0- graph. There are buttons in the Analysis Panel of the user’s interface, that show the result of each operation.

**The Code**

As usual, we use a class with a solution object and a main function, constructLMST, to implement the algorithm’s operations. Here constructLMST iterates through the nodes list and for each node it calculates its Local MST and pushes it to an array with all other LMSTs.

```
//Main Object
var lmstObject = function(){
   var that = this;
   that.solution = {
      "Topology" : [], //the final graph after the calculation of all LMSTs and the neighbor relations
      "LMSTs" : [], //list of all LMST trees, not the same as above (read algorithm)
      "step_data" : solutionFactory.newSolution(),
      "uni-directional" : []
   };
   that.constructLMST = function(network){
      var subnetwork;
      var edges;
      for(var i=0; i<network.nodes.length; i++){
         that.solution["step_data"].createStep();
         that.solution["step_data"].steps[i].text = "<p class=\"colored-text2\">Current node : "+network.nodes[i].id+".</p>";
         subnetwork = _returnSubnetwork(network.nodes[i], network);
         edges = _returnEdges(subnetwork, that.solution["step_data"]);
         that.solution["LMSTs"].push({ id : network.nodes[i].id , tree: _executePrim(subnetwork, edges,
that.solution["step_data"])});
      }
      that.solution["uni-directional"] = _uniDirectionalEdges(that.solution["LMSTs"], that.solution["Topology"]);
      return that.solution;
   };
}
```

_returnSubnetwork returns the subgraph (subnetwork) induced by the network graph, with only the node and its neighbors and their connections. To record all subnetwork edges and calculate their distances, we use _returnEdges. Then _executePrim constructs the LMST with the edges as input, along with the subgraph from _returnSubnetwork.

_chooseEdgeFunction below, is called each time to select an edge based on the calculation of weights, as mentioned in Topology Construction.

```
//The function that Prim's Algorithm will use to choose the next edge
var _chooseEdgeFunction = function(edges, step_data){
    var result;
    var minDistanceEdges = _minimumDistanceEdges(edges);
    if(minDistanceEdges.length == 1){ //only one edge has the minimum distance, we can stop
        result = minDistanceEdges[0];
    }
    else{ //more than one edges have the minimum distance
        //get the ones with max(vertex id)
        step_data.steps[step_data.steps.length - 1].text += "<p>Multiple edges have the same minimum distance
:</br>"+_stringifyEdgeList(minDistanceEdges)
        +"We calculate the value of max(source_id, target_id) as the 'weight' for all of them. Then we keep the one\
         (or those) with the minimum such value.</p>";
        var maxEdges = _maxVertexIdEdges(minDistanceEdges);
        if(maxEdges.length == 1){   //only one edge with the minimum value, i'm done
            result = maxEdges[0];
        }
        else{   //still more than one edges have the minimum value
            //from the edges with the same max values get the one with the minimum(source_id, target_id)
            step_data.steps[step_data.steps.length - 1].text += "<p>Still more than one edges with the previous minimum value
:</br>"
            +_stringifyEdgeList(maxEdges) + "Now for these ones we"+
            " calculate the minimum(source_id, target_id) and get the unique one with the smallest such value.</p>";
            result = _minVertexIdEdges(maxEdges);
        }
    }
    return result;
}
```

_executePrim, in the next page, has an array called availableEdges, in which we keep the available edges to choose from at every step of Prim’s execution. The verticesLeft array keeps which vertices haven’t been visited yet and is used for checking when to stop the calculations. The verticesVisited array keeps which vertices we have visited, to help recalculate the available edges to choose from, at each Prim step. The edgeSelected variable holds the result of  _chooseEdgeFunction mentioned above and edgesUsed holds all the edges selected at the end of the execution.

```
//Excute Prim's Algorithm on the given subnetwork
var _executePrim = function(subnetwork, edges, step_data){
   //keep a list with all the vertices to visit except for the 1st node
   step_data.steps[step_data.steps.length - 1].text += "<p class=\"colored-text2\">Prim's execution for this subnetwork : </p>";
   var verticesLeft = [];
   for(var i=1; i<subnetwork.nodes.length; i++){
      verticesLeft.push(subnetwork.nodes[i].id);
   }
   //Start from the root node
   var availableEdges = [];
   for(var i=0; i<subnetwork.nodes[0].edges.length; i++){
      availableEdges.push(edges[ subnetwork.nodes[0].edges[i] ]);
   }
   var verticesVisited = [subnetwork.nodes[0].id];
   step_data.steps[step_data.steps.length - 1].text += "<p class=\"colored-text3\">Vertices visited so far : ["+verticesVisited+"]</p>";
   step_data.steps[step_data.steps.length - 1].text += "<p>Choosing edge from :</br>"+_stringifyEdgeList(availableEdges)+"</p>";
   var edgesUsed = [];
   var edgeSelected = _chooseEdgeFunction(availableEdges, step_data); //choose an edge
   step_data.steps[step_data.steps.length - 1].text += "<p class=\"colored-text2\">Selected edge [ source :
"+edgeSelected["source"]+" , target : "+edgeSelected["target"]
   +", distance : "+edgeSelected["distance"]+" ]</p>";
   edgesUsed.push(edgeSelected);
   var vertexVisited = _returnOtherVertex(subnetwork.nodes[0].id, edgeSelected); //keep the vertex visited following this edge
   step_data.steps[step_data.steps.length - 1].text += "<p class=\"colored-text3\">New vertex added : "+vertexVisited+"</p>";
   verticesLeft = verticesLeft.filter(function(el){ //remove the visited vertex from the remaining vertices to visit
      return el != vertexVisited;
   });
   verticesVisited.push(vertexVisited); //keep an array with all the vertices visited so far
   //Keep running Prim until all vertices are visited
   var node;
   var condition1;
   var condition2;
   while((verticesLeft.length != 0)){
      availableEdges = [];
      for(var i=0; i<verticesVisited.length; i++){ //get all available edges from the vertices visited so far
         node = netOperator.returnNodeById(verticesVisited[i], subnetwork);
         for(var j=0; j<node.edges.length; j++){
            //if we haven't visited a vertex of this edge (this node is either source or target of the edge)
            condition1 = _.indexOf(verticesVisited, edges[ node.edges[j] ]["source"]) != -1;   //source visited
            condition2 = _.indexOf(verticesVisited, edges[ node.edges[j] ]["target"]) != -1;   //target visited
            if(!(condition1 && condition2)){
               availableEdges.push(edges[node.edges[j]]);
            }
         }
      }
      step_data.steps[step_data.steps.length - 1].text += "<p class=\"colored-text3\">Vertices visited so far :
["+verticesVisited+"]</p>";
      step_data.steps[step_data.steps.length - 1].text += "<p>Choosing edge from
:</br>"+_stringifyEdgeList(availableEdges)+"</p>";
      var edgeSelected = _chooseEdgeFunction(availableEdges, step_data); //choose an edge
      step_data.steps[step_data.steps.length - 1].text += "<p class=\"colored-text2\">Selected edge [ source :
"+edgeSelected["source"]+" , target : "+edgeSelected["target"]
   +", distance : "+edgeSelected["distance"]+" ]</p>";
      edgesUsed.push(edgeSelected);
      var vertexVisited = _returnNewVertex(verticesVisited, edgeSelected); //keep the vertex visited following this edge
      step_data.steps[step_data.steps.length - 1].text += "<p class=\"colored-text3\">New vertex added : "+vertexVisited+"</p>";
      verticesLeft = verticesLeft.filter(function(el){ //remove the visited vertex from the remaining vertices to visit
         return el != vertexVisited;
      });
      verticesVisited.push(vertexVisited);
   }
   return edgesUsed;
}
```

## Relative Neighborhood Graph (RNG)

Belongs in the topology solution category. An edge between two nodes A and B of the network graph is included in the RNG if there does not exist another node W inside the intersection of the circles formed by the transmission range of A and B. That means that in RNG, edge A-B won't be included if, there is a common neighbor of A and B, W, and the edges W-B and W-A have shorter distance than edge A-B. In real applications RNG can be deduced locally by each node using only the distance with its neighbors and exchanging messages including the coordinates to figure out those distances. In this application we skip the messaging phase, which is not a main part of this algorithm and use directly the knowledge of the positions of the nodes in the graph view that the user sees.

### The Code

This class has a solution field containing the final RNG and a "step-data" object, which records the solution's steps' data. The method performing the main implementation is constructRNG, which for each node of the network, calls the _calculateLocalRNG function.

```
var rngObject = function(){
    var that = this;
    that.solution = {
        "RNG" : [],
        "step_data" : solutionFactory.newSolution()
    };
    that.constructRNG = function(network){
        for(var i=0; i<network.nodes.length; i++){
            that.solution["RNG"].push(_calculateLocalRNG(network.nodes[i], network, that.solution["step_data"]));
        }
        return that.solution;
    };
}
```

_calculateLocalRNG takes as input a specific node, let’s call it A, and the network graph and it calculates the distances of A with its neighbors. Then for each neighbor B, it takes its distance with any other neighbor of A, which we’ll call C, as long as C is connected with B. If that distance is smaller than the distance of the A-B edge and A-C has also smaller distance from A-B, then A-B cannot be included in the RNG. Otherwise it is pushed to the “edges” array that is returned at the end.

```
var _calculateLocalRNG = function(node, network, solution){
    var neighbors;
    var distances = []; //distance from node to each neighbor
    var distance;
    var x_diff;
    var y_diff;
    neighbors = netOperator.returnNeighborObjects(node, network);
    solution.createStep();
    solution.steps[solution.steps.length - 1].text = "<p class=\"colored-text2\">Current node "+node.id+".</p>";
    //First calculate the distances of node from all its neighbors
    for(var i=0; i<neighbors.length; i++){
        x_diff = node.position.x - neighbors[i].position.x;
        y_diff = node.position.y - neighbors[i].position.y;
        distance = Math.sqrt(Math.pow(x_diff, 2) + Math.pow(y_diff, 2));
        distances.push(distance);
    }
    //Then for each neighbor check if their link with the node should be in the RNG.
    var edges = [];
    var valid;
    for(var i=0; i<neighbors.length; i++){
        valid = true;
        for(var j=0; j<neighbors.length; j++){  //is there any other neighbor inside the intersection of the 2 nodes' circles?
            if((i != j) && (_.indexOf(neighbors[i].neighbors, neighbors[j].id) != -1)){
                x_diff = neighbors[i].position.x - neighbors[j].position.x;
                y_diff = neighbors[i].position.y - neighbors[j].position.y;
                distance = Math.sqrt(Math.pow(x_diff, 2) + Math.pow(y_diff, 2)); //distance between the 2 neighbors
                if((distances[j] < distances[i]) && (distance < distances[i])){
                    valid = false;
                    solution.steps[solution.steps.length - 1].text += "<p>Edge between "+node.id+" and "+neighbors[i].id+
                    " omitted because of neighbor node "+neighbors[j].id+" between them.</p>";
                    break;
                }
            }
        }
        if(valid){
            edges.push( {"source" : node.id, "target" : neighbors[i].id });
            solution.steps[solution.steps.length - 1].text += "<p>Selected edge ["+node.id+" , "+neighbors[i].id+"].</p>";
        }
    }
    return edges;
}
```

## Gabriel Graph (GG)

Very similar to the RNG algorithm. The only change is that to include an edge between a node A and a node B in the GG, no common neighbor should exist inside the circle created with center the center of the A-B edge line and diameter the A-B line length.

### The Code

We have the ggObject class, that defines a solution object field containing everything recorded during execution, and the constructGG function that iterates through each node and calculates which edges will be in the final Gabriel Graph.

_calculateLocalGG calculates the distance between an input node A and all its neighbors, along with the position of the centers and the radii of the circles mentioned above. Then for each neighbor B, it takes any other neighbor C of A, that is connected to B and calculates the distance of this neighbor from the center of the circle corresponding to the line between A and B. If that distance is smaller than the corresponding radius, it doesn’t add edge A-B to the GG.

```
var ggObject = function(){
   var that = this;
   that.solution = {
      "GG" : [],
      "step_data" : solutionFactory.newSolution()
   };
   that.constructGG = function(network){
      for(var i=0; i<network.nodes.length; i++){
         that.solution["GG"].push(_calculateLocalGG(network.nodes[i], network, that.solution["step_data"]));
      }
      return that.solution;
   };
}

var _calculateLocalGG = function(node, network, solution){
   var neighbors = netOperator.returnNeighborObjects(node, network);
   var distance;
   var centers = []; //center point of the distance between node and each neighbor
   var radii = []; //radius of circle with diameter the line between node's position and a neighbor's position
   var x_diff;
   var y_diff;
   solution.createStep();
   solution.steps[solution.steps.length - 1].text = "<p class=\"colored-text2\">Current node "+node.id+".</p>";
   //First calculate the distances of node from all its neighbors
   for(var i=0; i<neighbors.length; i++){
      x_diff = node.position.x - neighbors[i].position.x;
      y_diff = node.position.y - neighbors[i].position.y;
      distance = Math.sqrt(Math.pow(x_diff, 2) + Math.pow(y_diff, 2));
      centers.push({ x : (node.position.x - x_diff/2), y : (node.position.y - y_diff/2)});
      radii.push(distance/2);
   }
   //Now add the apropriate edges to the GG
   var edges = [];
   var valid;
   for(var i=0; i<neighbors.length; i++){
      valid = true;
      for(var j=0; j<neighbors.length; j++){
         if((i != j) && (_.indexOf(neighbors[i].neighbors, neighbors[j].id) != -1)){
            x_diff = centers[i].x - neighbors[j].position.x;
            y_diff = centers[i].y - neighbors[j].position.y;
            distance = Math.sqrt(Math.pow(x_diff, 2) + Math.pow(y_diff, 2));
            if(distance < radii[i]){
               valid = false;
               solution.steps[solution.steps.length - 1].text += "<p>Edge between "+node.id+" and "+neighbors[i].id+
               " omitted because of neighbor node "+neighbors[j].id+" between them.</p>";
               break;
            }
         }
      }
      if(valid){
         edges.push( {"source" : node.id, "target" : neighbors[i].id });
         solution.steps[solution.steps.length - 1].text += "<p>Selected edge ["+node.id+" , "+neighbors[i].id+"].</p>";
      }
   }
   return edges;
}
```

# Chapter 3 – Users And Sessions

## Databases

The MySQL databases and their tables can be created by simply running /Databases/create_db.sql. In the “sessions” database no tables are created by this script, because the mysql session store module handles this from within the app.

## Users

In this section, it will be described how the user’s database, called by default “adhoced”, is defined.

There is a table Users, keeping the first name, the last name, the username and the password for each user. There is also an “id” column as a Primary Key. The first name is displayed at the workspace when a user is logged in.

There are two more tables, Networks and Nodes. Networks has  Primary Key “id” too,  a “name” column, which refers to the text the user entered for this network graph’s name and a Foreign Key “user_id”, which points to the “id” value in the Users table. That way each network belongs to a user.

The Nodes table has a Primary Key “id” as well, a Foreign Key “network_id” that points to the network that it belongs to, a “node_id” number indicating the id of the node in the user’s graph, a column “neighbors” which contains the neighbors of that node in the user’s graph in a parsable string form and finally “position_x” and “position_y” keeping the node’s position in the user interface when the save operation was issued.

Every available operation that the server wishes to perform, along with the configuration to the “adhoced” database is located in /Database/queries.js. The users’ database uses a connection pool, to reuse released connections instead of creating new ones each time. The default maximum is 200. Not all maximum connections are created at once. The description in the mysql module says that a new connection will only be created if there is no idle connection at the pool to handle the request at that time. Otherwise a previously released one will handle the task. It should also be noted that Node.js will perform these operations asynchronously, since it supports non-blocking I/O, meaning that the rest of the code will continue executing and the application will continue handling requests, no matter if these operations have finished. That is the reason that the response object is passed as a parameter to the functions that perform database operations, so whenever they complete, the response will be sent to the client by them.

## Sessions

The session store module contains functions that perform the database operations needed to keep the sessions database updated. So in the application there is no need to write extra code to manage this database. To identify a user that performed a request, we simply ask for a session to be retrieved if it exists, by giving to the store the session id inside the request’s body. We also leave it to the store to update the database when a session is created or destroyed, we just use the commands supported by the module to ask for such operations. The /Database/sessions.js module has the configurations and the session store object that is exported and available to other modules that wish to manage sessions.

# Chapter 4 – Adding More Features

This software's code is organized in a specific folder hierarchy, to make maintenance and adding extra features easier by anyone other than the writer of this Thesis. As mentioned in the above sections, the project is available on GitHub, meaning the source is open for anyone who wishes to set it up on their own machines and change it according to their needs. In this section, one can read which files they need to change, in order to add more features in specific parts of the app.

## Add User Interface Features

The main files to look at, when attempting this, are :

- /Client/Views/workspace.hbs (and maybe the no-session version workspaceNoSession.hbs)
- /Client/JavaScript/workspaceNetworkManager.js
- /Client/JavaScript/workspaceViewManager.js (the workspaceViewNoSession.js might need an update too)
- /Client/JavaScript/workspaceResultView.js
- /Client/JavaScript/External/joint_mod.js

All of them contain some global variables accessible by the others, when needed.

In workspace.hbs, we have the HTML defining the layout of the user's workspace. Adding buttons or other elements can be done here.

The file workspaceNetworkManager.js contains the functionality related to the network graph in the UI and the tools that are used to edit this graph.

The view of the page, such as any operations taking place when resizing the window or hiding and showing dialogues is handled in workspaceViewManager.js. Also the functionality of the load and save buttons, including the resizing of the whole graph to fit the Graph View's window is handled in this file. Finally, all functions that manage the data inserted in the algorithm dialogues that open when the "Execute" button is pressed and the Ajax calls to the server, are declared here.

The workspaceResultView.js file contains functions that handle how the client's page responds visually after the result of the execution of the algorithm is sent by the server. For every algorithm, there is a function that changes the text of the Analysis Panel, to depict the final results or a step's result of the execution. All graph repainting to depict results is handled here too.

The Joint.js extension was used to make the graphs. Note that joint_mod.js is a modification of joint.js inside the same folder, that contains a few changes of the functionality behind the Graph View.

## Add More Algorithms

1) To add more algorithms, one must first create a module for that algorithm solution, preferably inside the /Server/Algorithms folder. Reading some of the other algorithm modules can give an insight as to how to organize the solution. Use the classes inside /Server/Algorithms/steps.js, to record the execution steps. Use the /Server/Algorithms/networkOperations.js module when you need to retrieve data from the network object given by the user. Here we use a class named after the algorithm's name, with fields containing the data needed for the solution and a main method that is called to execute the solution, given the graph and maybe extra parameters as input. The usage of classes with their individual data is necessary, since in Node.js any global variables inside a module will be shared and editable by all who access the module. For instance if two users execute the same algorithm at the same time and a global variable "solution" is declared that will keep the result of the execution, they will both access and edit the same variable, the requests won't have their own different instance of the module. So by using classes, for each request we create an object of the class we need and that object keeps the individual data for that request. These modules export only a function called "Factory", which returns a new object of the main class in the module.
2) The next thing to do is to change the "handler" object inside /Server/Routers/workspaceRouter.js and specifically include the new algorithm code in "routeRequest" and write a function like the other ones inside the "handler" object, to request a solution by the module you wrote and return the response to the user. Each algorithm written so far has a code that is sent to the server, so it can execute the corresponding function. By convention, the name is "alg_" + (number of algorithm), meaning the first algorithm is "alg_1", the second "alg_2" and so on. This name is also the id of the buttons in the UI.
3) After step 2, one can create the buttons for the algorithms, the dialogues for the extra data needed by the user and the Ajax calls to the server, by changing :
    a. /Client/Views/workspace.hbs (and maybe the no-session version workspaceNoSession.hbs)

b. /Client/JavaScript/workspaceViewManager.js (the workspaceViewNoSession.js might need an update too)

In the latter file, the ajaxObject is the one sent to the server as JSON. The code field contains the id of the button that was pressed, which is “alg_1”, “alg_2” and so on, as described by the previous step.

4) When the user interface includes the new buttons and the Ajax calls are written, the final step is to add a function inside /Client/JavaScript/workspaceResultView.js to make any visual changes to the graph after the execution and fill the Analysis Panel with the text of the solution. There are many variables declared at the top of the file, which can store data after the execution or color styling information for painting the graph. The most used one is the stepDataArray, which keeps the data of each step in a list, to be able to use that data when a step box in the analysis panel is clicked, to paint the graph to represent the results of that step. One must also change handleResponse, which is called after an Ajax response to an algorithm request, to include the function they’ve written.

# Chapter 5 – References

## Algorithm Implementation Based On :

- Wu & Li algorithm : **"On Calculating Connected Dominating Set for Efficient Routing in Ad Hoc Wireless Networks"** by Jie Wu and Hailan Li
- Multipoint Relays : **"Computing connected dominated sets with multipoint relays"** by Cedric Adjih, Philippe Jacquet, Laurent Viennot
- DCA clusters : **"On the Complexity of Clustering Multi-Hop Wireless Networks"** by Stefano Basagni
- Max-Min D-Cluster : **"Max-Min D-Cluster Formation in Wireless Ad Hoc Networks"** by Alan D. Amis, Ravi Prakash, Thai H.P. Vuong, Dung T. Huynh and **"On multihop clusters in wireless sensor networks"** by Alexandre Delye, Michel Marot and Monique Becker
- Maximal Independent Set : **"Distributed Construction of Connected Dominating Set in Wireless Ad Hoc Networks"** by Peng-Jun Wan, Khaled M. Alzoubi, Ophir Frieder and **"Propagation and Leader Election in a Multihop Broadcast Environment"** by Israel Cidon and Osnat Mokryn
- LMST : **"Design and Analysis of an MST-Based Topology Control Algorithm"** by Ning Li, Jennifer C. Hou, Lui Sha
- RNG : **"Localized LMST and RNG based minimum-energy broadcast protocols in ad hoc networks"** by Julien Cartigny, Francois Ingelrest, David Simplot-Ryl, Ivan Stojmenovic
- GG : **"Distributed Topology Control in Wireless Ad Hoc Networks using β-Skeletons"** by Manvendu Bhardwaj, Satyajayant Misra and Guoliang Xue