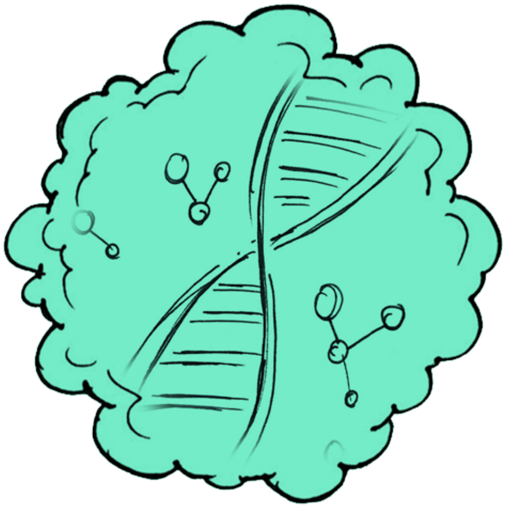
# Algorithmes génétiques pour l'intelligence artificielle - Annexe 1 - Code source



Thèse de bachelor présentée par

**M. Thomas Ibanez**

Pour l'obtention du titre Bachelor of Science HES-SO en

**Ingénierie des technologies de l'information avec orientation en Logiciels et Systèmes complexes**

Professeur HES responsable
**Orestis Malaspinas**

**Septembre 2018**

# Table des matières

# 1 Serveur

server.js

```
/**
 * This module handles web-client requests
 *
 * @author Thomas Ibanez
 * @version 1.0
 */
"use strict";
const proto = require(’./protobuf/mg_pb.js’);
const mgnetwork = require(’./mgnetwork.js’)
const mgpool = require(’./pool.js’);
const mgNEAT = require(’./neat.js’)
const genetic = require(’./genetic.js’);
const express = require(’express’);
const cors = require(’cors’);
const log = require(’winston’);
const mongoose = require(’mongoose’);
const app = express();
const bodyParser = require(’body-parser’);
const pool = new mgpool.Pool(500);

log.level = ’debug’;
app.use(cors());
app.use(bodyParser.urlencoded({extended : false}));

mongoose.connect(’mongodb://localhost/mg’);
let db = mongoose.connection;
let games = [
    {
        id : 0,
        name : "Asteroid"
    }
];
let topos = [
    {
        gameId : 0,
        netType : proto.MGNetworkType.MG_MULTILAYER_PERCEPTRON,
        netMetadata : {
            inputCount : 8,
            hLayerCount : 2,
            hLayers : [12, 8],
            outputCount : 4
        }
    },
    {
        gameId : 0,
        netType : proto.MGNetworkType.MG_MULTILAYER_PERCEPTRON,
        netMetadata : {
            inputCount : 8,
            hLayerCount : 1,
            hLayers : [12],
            outputCount : 4
        }
    },
    {
        gameId : 0,
        netType : proto.MGNetworkType.MG_NEAT,
        netMetadata : {
            inputCount : 8,
            hLayerCount : 0,
            outputCount : 4
        }
    }
];
db.on(’error’, console.error.bind(console, ’connection error:’));

let genSchema = mongoose.Schema({
    batchId : Number,
    genNumber : Number,
    topoID : Number,
    avgFitnesses : [Number],
    bestFitnesses : [Number],
    genomes : [String],
    species : [{
        bestFitness : Number,
        best : String,
        staleness : Number,
        averageFitness : Number
    }],
    innovationHistory : [{
        from : Number,
        to : Number,
        innovationNumber : Number,
        innovationNumbers : [Number]
    }]
});
let saved = [];
let Gen = mongoose.model(’Generation’, genSchema);
db.once(’open’, function() {
    Gen.find(function (err, gens) {
        saved = gens;
    });
});

app.set('view engine', 'pug');
app.set('view options', {"pretty" : true});
app.use('/scripts', express.static(__dirname + '/scripts/'));
app.locals.pretty = true;

mgnetwork.init(pool);

let topoID = 0;

app.get('/', (req, res) => {
  res.render('main', {
        "workers" : pool.workers,
        "games" : games,
        "currentGame" : pool.currentGame,
        "currentTopo" : topoID,
        "currentType" : pool.currentType,
        "remainingCycles" : pool.targetCycles - pool.cycles,
        "topologies" : topos,
        "saves" : saved
    });
});

app.get('/status', (req, res) => {
  res.json({
        "workers" : pool.workers,
        "games" : games,
        "currentGame" : pool.currentGame,
        "currentTopo" : topoID,
        "currentType" : pool.currentType,
        "currentGeneration" : pool.cycles,
        "remainingCycles" : isFinite(pool.targetCycles) ? pool.targetCycles -
pool.cycles  : "Infinity",
        "avgFitnesses" : pool.avgFitnesses.slice(pool.avgFitnesses.length < 50 ? 0  :
pool.avgFitnesses.length - 50, pool.avgFitnesses.length),
        "bestFitnesses" : pool.bestFitnesses.slice(pool.bestFitnesses.length < 50 ? 0
 : pool.bestFitnesses.length - 50, pool.bestFitnesses.length),
        "topologies" : topos
    });
});

app.get('/saves', (req, res) => {
  res.json({
        "saves" : saved
    });
});

app.post('/task', (req, res) => {
    if(req.body.pause != null) {
        pool.pauseTask();
    } else {
        let n = (req.body.infgen != null ? Infinity  : (req.body.onegen != null ? 1
 : 100));
        pool.newTask(n);
    }
    res.redirect("/");
});

app.post('/work', (req, res) => {
    if(req.body.lock) {
        if(pool.currentGame == "-") {
            pool.lockInfo(games[req.body.tgame].name, req.body.tnet,
topos[req.body.ttopo % 1000].netMetadata);
            topoID = req.body.ttopo;
        } else {
            pool.lockInfo(null, null, null);
        }
    } else if(req.body.save) {
        let current = new Gen({
            batchId : Math.floor((new Date).getTime() / 1000),
            genNumber : pool.cycles,
            topoID : topoID,
            avgFitnesses : pool.avgFitnesses,
            bestFitnesses : pool.bestFitnesses,
            genomes : pool.genomes.map(x => JSON.stringify(x)),
            species : mgNEAT.getSpecies().map(x => {return {bestFitness :
x.bestFitness, best : JSON.stringify(x.best), staleness : x.staleness,
averageFitness : x.averageFitness}}),
            innovationHistory : mgNEAT.getInnovationHistory()
        });
        current.save(function(err) {
            Gen.find(function (err, gens) {
                saved = gens;
            });
        });
    } else if(req.body.load) {
        Gen.findOne({ batchId : req.body.lbatch }).lean().exec(function(err, result) {
            pool.lockInfo(games[topos[result.topoID % 1000].gameId].name,
topos[result.topoID % 1000].netType, topos[result.topoID % 1000].netMetadata);
            topoID = result.topoID;
            pool.cycles = result.genNumber,
            pool.targetCycles = result.genNumber,
            pool.avgFitnesses = result.avgFitnesses,
            pool.bestFitnesses = result.bestFitnesses,
            pool.genomes = result.genomes.map(x => JSON.parse(x));
            mgNEAT.setSpecies(result.species.map(x =>
                {
                    let s = new mgNEAT.Specie(JSON.parse(x.best));
                    s.staleness = x.staleness ;
                    s.averageFitness = x.averageFitness ;
                    return s ;
                }
            ));
            mgNEAT.setInnovationHistory(result.innovationHistory);
        });
    } else if(req.body.regen) {
        pool.createInitialPopulation();
        pool.sendTasksToClients();
    }
    res.redirect("/");
});

app.listen(8080,  () => {
    log.info("Express running  ADDRESS http://localhost:8080");
});
```

pool.js

```
/**
 * This module handles the workers pool, and distributes the work
 *
 * @author Thomas Ibanez
 * @version 1.0
 */
"use strict" ;
const mgclient = require('./client.js’);
const mgnetwork = require('./mgnetwork.js’);
const proto = require('./protobuf/mg_pb.js’);
const genetic = require('./genetic.js’);

function Pool(population) {
    this.workers = {} ;
    this.spectators = {} ;
    this.species = [];
    this.genomes = [];
    this.avgFitnesses = [];
    this.bestFitnesses = [];
    this.cycles = 0 ;
    this.targetCycles = 0;
    this.population = population;
    this.currentGame = "-";
    this.currentType = proto.MGNetworkType.MG_MULTILAYER_PERCEPTRON;
    this.currentTopo = "";
    this.idle = true;
    this.computingGenomes = 0;
}

/**
 * Adds a worker to the pool
 * @param  {String} id   id of the worker
 * @param  {String} name Name of the worker
 */
Pool.prototype.addWorker = function(id, name) {
    this.workers[id] = new mgclient.Client(name);
    if( !this.idle) {
        this.sendTasksToClients();
    }
}

/**
 * Removes a worker from the pool
 * @param  {String} id id of the worker
 */
Pool.prototype.removeWorker = function(id) {
    if(this.workers[id] != undefined && this.workers[id].busy) {
        this.genomes[this.workers[id].genomeID].computing = false;
        this.genomes[this.workers[id].genomeID].waiting = false;
    }
    delete this.workers[id];
    delete this.spectators[id];
}

/**
 * Adds a spectator
 * @param  {[type]} id   Id of the spectator
 * @param  {[type]} name Name of the spectator
 */
Pool.prototype.addSpectator = function(id, name) {
    this.spectators[id] = new mgclient.Client(name);
}

/**
 * Initalizes the pool with a random population made for the selected task
 */
```

```
Pool.prototype.createInitialPopulation = function() {
    this.genomes = genetic.createRandomGeneration(this.currentType, this.population,
this.currentTopo);
}

/**
 * Launches a new task for a given number of generations
 * @param  {Number} numGens Number of generations to make, you can use Infinity
 */
Pool.prototype.newTask = function(numGens) {
    if( !this.idle ||ăthis.currentGame == "-")
        return;
    this.targetCycles = this.cycles + numGens;
    this.idle = false;
    this.computingGenomes = 0;
    this.sendTasksToClients();
};

/**
 * Lock the selected task infos
 * @param  {String} game Game to play
 * @param  {Number} type Network type
 * @param  {Object} topo Network topology
 */
Pool.prototype.lockInfo = function(game, type, topo) {
    if( !this.idle)
        return;
    if(this.currentGame != "-") {
        this.currentGame = "-";
    } else {
        this.currentGame = game;
        this.currentType = type;
        this.currentTopo = topo;
        this.avgFitnesses = [];
        this.bestFitnesses = [];
        this.cycles = 0;
        this.targetCycles = 0;
        this.createInitialPopulation();
    }
};

/**
 * Response callback
 * @param  {String} id      Worker ID
 * @param  {MGComputeResponse} message Message content
 */
```

```
Pool.prototype.onResponse = function(id, message) {
    if(message.getCanDo() != true) {
        this.workers[id].status = "Unable to compute" ;
        this.workers[id].busy = false ;
    } else {
        this.workers[id].status = "Computing..." ;
        this.workers[id].busy = true ;
        this.genomes[this.workers[id].genomeID].computing = true ;
    }
    this.genomes[this.workers[id].genomeID].waiting = false ;
}

/**
 * Result callback
 * @param  {String} id       Worker ID
 * @param  {MGComputeResult} message Message content
 */
Pool.prototype.onResult = function(id, message) {
    this.workers[id].status = "Waiting..." ;
    this.workers[id].busy = false ;
    this.genomes[this.workers[id].genomeID].fitness = message.getFitness();
    this.genomes[this.workers[id].genomeID].unajustedFitness = message.getFitness();
    this.sendTasksToClients();
}

/**
 * Distributes work to clients
 */
Pool.prototype.sendTasksToClients = function() {
    let allDone = false ;

    for (let index in this.workers) {
        let w = this.workers[index];
        if(w.busy == false) {
            allDone = true ;
            for (let i = 0 ; i < this.genomes.length ; i++) {
                if(this.genomes[i].fitness == -1) {
                    allDone = false ;
                }
                if(this.genomes[i].computing == false && !this.genomes[i].waiting) {
                    allDone = false ;
                    this.genomes[i].waiting = true ;
                    w.genomeID = i ;
                    let computeInfo = new proto.MGComputeInfo();
                    computeInfo.setGame(this.currentGame);
                    computeInfo.setNetType(this.currentType);
computeInfo.setNetMetadata(genetic.metadataFromTopology(this.currentTopo));
                    let request = new proto.MGComputeRequest();
                    request.setComputeInfo(computeInfo);
                    request.setGenome(genetic.genomeString(this.genomes[i],
this.currentType));
                    mgnetwork.sendTo(index, proto.MGMessages.MG_COMPUTE_REQUEST,
request);
                    w.busy = true;
                    break;
                }
            }
        }
    }

    if(allDone && !this.idle) {
        this.cycles++;
        this.idle = (this.cycles == this.targetCycles);
        //Compute generation's average fitness
        this.avgFitnesses[this.cycles - 1] = this.genomes.map(x =>
x.unajustedFitness).reduce((a,c) => a + c) / this.population;
        //Get best fitness
        this.bestFitnesses[this.cycles - 1] = this.genomes.map(x =>
x.unajustedFitness).reduce((a, c) => (a > c) ? a  : c);

        let best = this.genomes.reduce((a, c) => (a.unajustedFitness >
c.unajustedFitness) ? a  : c);
        let computeInfo = new proto.MGComputeInfo();
        computeInfo.setGame(this.currentGame);
        computeInfo.setNetType(this.currentType);
        computeInfo.setNetMetadata(genetic.metadataFromTopology(this.currentTopo));
        let request = new proto.MGComputeRequest();
        request.setComputeInfo(computeInfo);
        request.setGenome(genetic.genomeString(best, this.currentType));
        for (var a in this.spectators) {
            mgnetwork.sendTo(a, proto.MGMessages.MG_COMPUTE_REQUEST, request);
        }

        //Regen genomes
        this.genomes = genetic.createNextGeneration(this.genomes, this.currentType);
        //Reset client state
        for (let index in this.workers) {
            this.workers[index].busy = false;
        }
        if(!this.idle) {
            this.sendTasksToClients();
        }
    }
}

/**
 * Pauses the current task
 */
Pool.prototype.pauseTask = function() {
    this.targetCycles = this.cycles + 1;
    this.sendTasksToClients();
};

module.exports.Pool = Pool;
```

genetic.js

```
/**
 * This module is an interface to execute genetic algorithms
 *
 * @author Thomas Ibanez
 * @version 1.0
 */
"use strict";
const proto = require('./protobuf/mg_pb.js');
const NEAT = require('./neat.js');
const MLP = require('./mlp.js');

/**
 * Gives the metadata string to send to the client for a network topology
 * @param  {Object} topo The topology of the network
 * @return {String}      String representation of the topology
 */
function metadataFromTopology(topo) {
    return
topo.inputCount+","+topo.hLayerCount+","+topo.hLayers+","+topo.outputCount;
}

/**
 * Encodes the genome to a string
 * @param  {Object} genome The genome to encode
 * @param  {MGNetworkType} type   Enum value of the type of the genome
 * @return {String}         String representation of the genome
 */
function genomeString(genome, type) {
    if(type == proto.MGNetworkType.MG_MULTILAYER_PERCEPTRON) {
        return genome.code;
    } else {
        let geneCount = genome.genes.filter(g => g.enabled).length;
        let code = geneCount + "," + genome.nodes.length + "," + genome.biasNode + ","
+ genome.layers + "," ;
        for(let i in genome.genes) {
            if(genome.genes[i].enabled) {
                code += genome.genes[i].from + "," + genome.genes[i].to + "," +
genome.genes[i].weight + "," ;
            }
        }
        for(let i in genome.nodes) {
            code += genome.nodes[i].no + "," + genome.nodes[i].layer + (i ==
genome.nodes.length - 1 ? ""  : ",");
        }
        return code ;
    }
}

/**
 * Creates a random population
 * @param  {MGNetworkType} genomeType  The type of the genomes to create
 * @param  {Number} population  The amount of genome to create
 * @param  {Object} netMetadata The metadata of the genomes to create
 * @return {Array}              Array containing the random genomes
 */
function createRandomGeneration(genomeType, population, netMetadata) {
    let genomes = [];

    if(genomeType == proto.MGNetworkType.MG_MULTILAYER_PERCEPTRON) {
        let linkCount = (netMetadata.inputCount + 1) * netMetadata.hLayers[0];
        for (let k = 1 ; k < netMetadata.hLayerCount ; k++) {
            linkCount += netMetadata.hLayers[k] * netMetadata.hLayers[k-1];
        }
        linkCount += netMetadata.hLayers[netMetadata.hLayerCount - 1] *
netMetadata.outputCount ;

        for(let i = 0 ; i < population ; i++) {
            let genomeCode = "" ;
            for(let j = 0 ; j < linkCount ; j++) {
                genomeCode += (Math.random() * 2 - 1) + (j == linkCount - 1 ? "" :
",");
            }
            genomes.push({code : genomeCode, computing : false, fitness : -1});
        }
    } else {
        NEAT.setSpecies([]);
        NEAT.setInnovationHistory([]);
        for(let i = 0 ; i < population ; i++) {
            let g = {
                genes : [],
                nodes : [],
                inputs : netMetadata.inputCount,
                outputs : netMetadata.outputCount,
                layers : 2,
                nextNode : 0,
                biasNode : 0,
                computing : false,
                fitness : -1
            }
            //Inputs
            for (let i = 0 ; i < g.inputs ; i++) {
                g.nextNode++ ;
                g.nodes.push({no : i, layer : 0});
            }
            //Outputs
            for (let i = 0 ; i < g.outputs ; i++) {
                g.nextNode++ ;
                g.nodes.push({no : i + g.inputs, layer : 1});
            }
            //Bias
            g.nodes.push({no : g.nextNode, layer : 0});
            g.biasNode = g.nextNode ;
            g.nextNode++ ;

            //Connect inputs to outputs
            let next = 0 ;
            for (let i = 0 ; i < g.inputs ; i++) {
                for (let j = 0 ; j < g.outputs ; j++) {
                    g.genes.push({from : i, to : g.inputs + j, weight : Math.random()
* 2 - 1, innovationNo : next, enabled : true});
                    next++ ;
                }
            }

            //Connect bias to outputs
            for (let i = 0 ; i < g.outputs ; i++) {
                g.genes.push({from : g.biasNode, to : g.inputs + i, weight :
Math.random() * 2 - 1, innovationNo : next, enabled : true});
                next++ ;
            }
            genomes.push(g);
        }
    }
    return genomes ;
```

```
}

/**
 * Creates the next generation of genomes, using the previous generation which has
been evaluated
 * @param  {Array} genomes    The previous generation
 * @param  {MGNetworkType} genomeType The type of the genomes
 * @return {Array}            The next generation, using the right genetic algorithm
 */
function createNextGeneration(genomes, genomeType) {
    let nextgen = [];
    genomes.sort(function(a, b) { //Sort greater fitness first
        if(a.fitness < b.fitness) {
            return 1 ;
        } else if(a.fitness > b.fitness) {
            return -1 ;
        }
        return 0 ;
    });
    if(genomeType == proto.MGNetworkType.MG_MULTILAYER_PERCEPTRON) {
        nextgen = MLP.createNextGeneration(genomes);
    } else {
        nextgen = NEAT.createNextGeneration(genomes);
    }
    return nextgen ;
}

/**
 * Selects randomly a genome by taking in consideration it's fitness (CDF)
 * @param  {Array} population Whole population of genomes
 * @return {Genome}             The selected genome
 */
function select(population) {
    let fitsum = population.map(x => x.fitness).reduce((a,c) => a + c);
    let threshold = Math.random() * fitsum ;
    let sum = 0 ;
    for(let i in population) {
        sum += population[i].fitness ;
        if(sum >= threshold) {
            return population[i];
        }
    }
    return population[0];
}

module.exports.createRandomGeneration = createRandomGeneration ;
module.exports.metadataFromTopology = metadataFromTopology ;
module.exports.createNextGeneration = createNextGeneration ;
module.exports.genomeString = genomeString ;
module.exports.select = select ;
```

client.js

```
/**
 * This module is an abstract representation of a computing client
 *
 * @author Thomas Ibanez
 * @version 1.0
 */
"use strict" ;
function Client(wname) {
    this.name = wname ;
    this.status = "Waiting..." ;
    this.busy = false ;
    this.genomeID = -1 ;
}

module.exports.Client = Client ;
```

mgnetwork.js

```
/**
 * This module handles all the network communications for machine gaming
 *
 * @author Thomas Ibanez
 * @version 1.0
 */
"use strict" ;
const net = require(’net’);
const uuid = require("uuid/v1");
const log = require(’winston’);
const mgpool = require(’./pool.js’);
const mgproto = require(’./protobuf/mg_pb.js’);
const client = require(’./client.js’);

let connections = [];
let pool = null ;

/**
 * Handles the desire to join from a client
 * @param  {String} id      The unique client id
 * @param  {MGJoin} message The join message sent by the client
 */
function handleJoin(id, message) {
    let response = new proto.MGJoinResponse();
    //Always accepted
    response.setAccepted(true);
    sendTo(id, proto.MGMessages.MG_JOIN_RESPONSE, response);
    if(message.getSpectator() == true) {
        pool.addSpectator(id, message.getPrettyName());
    } else {
        pool.addWorker(id, message.getPrettyName());
    }
}

/**
 * Removes all of the resources used for a client
 * @param  {String} id ID of the client
 */
function dispose(id) {
    pool.removeWorker(id);
  if (connections[id] !== undefined)
    connections[id].destroy();
  delete connections[id];
}

/**
 * Initialize server socket to accept connections
 * @param  {Object} _pool The worker pool
 */
function init(_pool) {
    pool = _pool;
    net.createServer(function(sock) {
        let id = uuid();
        let joined = false;
        connections[id] = sock;
        log.verbose(‘New connection from ${id} (${sock.remoteAddress} :
${sock.remotePort})‘);

        sock.on("error", function(err) {
            dispose(id);
            console.log("Caught flash policy server socket error: ");
            console.log(err.stack);
        });

        sock.on(’data’, function(data) {
            let bytes = Array.prototype.slice.call(data, 0);
            let offset = 0;
            do {
                let type = bytes[offset];
                let size = bytes[offset + 1];
                let message = null;
                switch(type) {
                    case proto.MGMessages.MG_JOIN :
                        if(!joined) {
                            message =
proto.MGJoin.deserializeBinary(bytes.slice(offset + 2, offset + 2 + size));
                            handleJoin(id, message);
                            joined = true;
                        }
                        break;
                    case proto.MGMessages.MG_COMPUTE_RESPONSE :
                        message =
proto.MGComputeResponse.deserializeBinary(bytes.slice(offset + 2, offset + 2 + size));
                        pool.onResponse(id, message);
                        break;
                    case proto.MGMessages.MG_COMPUTE_RESULT :
                        message =
proto.MGComputeResult.deserializeBinary(bytes.slice(offset + 2, offset + 2 + size));
                        pool.onResult(id, message);
                        break;
                    case proto.MGMessages.MG_END :
                        message = proto.MGEnd.deserializeBinary(bytes.slice(offset +
2, offset + 2 + size));
                        break;

                 }
                 offset += size + 2;
             } while(offset < bytes.length);
        });

        sock.on('close', function(data) {
            log.verbose(`We received a close from ${id}`);
            dispose(id);
        });

    }).listen('4567', '127.0.0.1');
    log.info("WebSocket server is alive on port 4567");
}

/**
 * Sends a message to a client
 * @param  {String} id          Id of the client
 * @param  {Number} messageType Type of the message to send
 * @param  {Object} message     Object of the message made with protobuf
 */
function sendTo(id, messageType, message) {
    const buf = Buffer.alloc(5, 0);
    buf[0] = messageType;
    const mArray = message.serializeBinary();
    buf.writeUInt32BE(mArray.length, 1);
    if(connections[id] !== undefined) {
        connections[id].write(buf);
        connections[id].write(Buffer.from(mArray));
    }
}

module.exports.sendTo = sendTo;
module.exports.init = init;
```

mlp.js

```
/**
 * This module handles all the MLP Genetic Algorithm
 *
 * @author Thomas Ibanez
 * @version 1.0
 */
"use strict";
const genetic = require("./genetic.js");
const MUTATION_RATE = 0.1;

/**
 * Creates a new generation of genomes from the previous one
 * @param  {Array} genomes genomes of the previous generation
 * @return {Array}         new generations of genomes
 */
function createNextGeneration(genomes) {
    let nextgen = [];
    nextgen.push({code : genomes[0].code, computing : false, fitness : -1});
    for(let i = 1; i < genomes.length; i++) {
        let g = null;
        if(i < genomes.length / 2) {
            g = {code : genetic.select(genomes).code, computing : false, fitness :
-1};
        } else {
            g = {code : crossover(genetic.select(genomes).code,
genetic.select(genomes).code), computing : false, fitness : -1};
        }
        mutate(g, MUTATION_RATE);
        nextgen.push(g);
    }
    return nextgen;
}
```

```
/**
 * Makes an offspring from 2 parents by crossing over genes
 * @param  {Genome} g1 Parent 1
 * @param  {Genome} g2 Parent 2
 * @return {Genome}    Offspring
 */
function crossover(g1, g2) {
    let g1Array = g1.split(",");
    let g2Array = g2.split(",");
    let crosspoint = Math.round(Math.random() * (g1Array.length - 1));
    let newCode = "";
    for(let i = 0; i < g1Array.length; i++) {
        let g = "";
        if(i <= crosspoint) {
            g = g1Array[i];
        } else {
            g = g2Array[i];
        }
        newCode += g + (i == g1Array.length - 1 ? "" : ",");
    }
    return newCode;
}

/**
 * Mutates slightly the weight of a genome
 * @param  {Genome} g  genome to mutate
 * @param  {Number} mr probability of a mutation occuring
 */
function mutate(g, mr) {
    if(Math.random() < mr) {
        let gArray = g.code.split(",");
        let i = Math.ceil(Math.random() * (gArray.length - 1));
        let mutationGene = gArray[i];
        let newValue = parseFloat(mutationGene) + (Math.random() - 0.5);
        if(newValue > 1) {
            newValue = 1;
        } else if(newValue < -1) {
            newValue = -1;
        }
        gArray[i] = ""+newValue;
        g.code = "";
        for(let j = 0; j < gArray.length; j++) {
            g.code += gArray[j] + (j == gArray.length - 1 ? "" : ",");
        }
    }
}
```

```
module.exports.createNextGeneration = createNextGeneration ;
```

neat.js

```
/**
 * This module handles all the NEAT Algorithm
 *
 * @author Thomas Ibanez
 * @version 1.0
 */
"use strict" ;
const genetic = require("./genetic.js")

let nextConnectionNo = 1000 ;
let species = [];
let innovationHistory = [];

/**
 * Creates a new generation of genomes from the previous one
 * @param  {Array} genomes genomes of the previous generation
 * @return {Array}         new generations of genomes
 */
function createNextGeneration(genomes) {
    let nextgen = [];
    speciate(genomes);
    species.sort(function(a, b) { //Sort greater fitness first
        if(a.bestFitness < b.bestFitness) {
            return 1 ;
        } else if(a.bestFitness > b.bestFitness) {
            return -1 ;
        }
        return 0 ;
    });
    cullSpecies();
    killStaleSpecies();
    killBadSpecies(genomes.length);
    let averageSum = getAvgFitnessSum();
    for (let i in species) {
        nextgen.push(clone(species[i].genomes[0]));
        let childAlloc = Math.floor(species[i].averageFitness / averageSum *
genomes.length) - 1 ;
        for (let j = 0 ; j < childAlloc ; j++) {
            nextgen.push(species[i].yieldChild());
        }
    }
    for(let i = nextgen.length ; i < genomes.length ; i++) {
        nextgen.push(species[0].yieldChild());
    }
    for(let i in nextgen) {
        nextgen[i].fitness = -1 ;
        nextgen[i].computing = false ;
    }

    return nextgen ;
}

/**
 * Partition the genomes into theirs species
 * @param  {Array} genomes Genomes to classify
 */
function speciate(genomes) {
    for(let i in species) {
        species[i].clear();
    }
    for(let i in genomes) {
        let speciesFound = false ;
        for(let j in species) {
            if(species[j].sameSpecies(genomes[i])) {
                species[j].addToSpecies(genomes[i]);
                speciesFound = true ;
                break ;
            }
        }
        if( !speciesFound) {
            species.push(new Specie(genomes[i]));
        }
    }
    for(let i in species) {
        species[i].genomes.sort(function(a, b) { //Sort greater fitness first
            if(a.fitness < b.fitness) {
                return 1 ;
            } else if(a.fitness > b.fitness) {
                return -1 ;
            }
            return 0 ;
        });
    }
}

/**
 * Cull the bottom half of each species, also shares fitness
 */
```

```
function cullSpecies() {
    for (let i in species) {
        species[i].cull();
        species[i].fitnessSharing();
        species[i].setAverage();
    }
}

/**
 * Removes the species who's fitness hasn't improved in 15 generations
 */
function killStaleSpecies() {
    for (let i = 0 ; i < species.length ; i++) {
        if (species[i].staleness >= 15) {
            species.splice(i, 1);
            i-- ;
        } else {
            species[i].staleness++ ;
        }
    }
}

/**
 * Removes species that are too bad to be given a child
 * @param  {Number} population total population size
 */
function killBadSpecies(population) {
    let averageSum = getAvgFitnessSum();

    for(let i = 0 ; i < species.length ; i++) {
        if(species[i].averageFitness / averageSum * population < 1) {
            species.splice(i, 1);
            i-- ;
        }
    }
}

/**
 * Get the sum of each species average fitness
 * @return {Number} Sum of all average fitnesses
 */
function getAvgFitnessSum() {
    return species.map(x => x.averageFitness).reduce((a, c) => a + c);
}

/**
```

```
 * Creates an offspring from 2 parents
 * @param  {Genome} g1 Parent 1
 * @param  {Genome} g2 Parent 2
 * @return {Genome}    Offspring
 */
function crossover(g1, g2) {
    let child = {
        genes : [],
        nodes : [],
        inputs : g1.inputs,
        outputs : g1.outputs,
        layers : g1.layers,
        nextNode : g1.nextNode,
        biasNode : g1.biasNode,
        computing : false,
        fitness : -1
    };

    let childGenes = [];
    let enabledGenes = [];

    for(let i in g1.genes) {
        let enabled = true;

        let parent2gene = matchingGene(g2, g1.genes[i].innovationNo);
        if (parent2gene != -1) {
            if (!g1.genes[i].enabled || !g2.genes[parent2gene].enabled) {
                if (Math.random() < 0.75) {
                    enabled = false;
                }
            }
            if (Math.random() < 0.5) {
                childGenes.push(g1.genes[i]);
            } else {
                childGenes.push(g2.genes[parent2gene]);
            }
        } else {
            childGenes.push(g1.genes[i]);
            enabled = g1.genes[i].enabled;
        }
        enabledGenes.push(enabled);
    }

    for (let i in g1.nodes) {
        child.nodes.push({no : g1.nodes[i].no, layer : g1.nodes[i].layer});
    }
    for(let i in childGenes) {
        child.genes.push({from : childGenes[i].from, to : childGenes[i].to, weight :
childGenes[i].weight, innovationNo : childGenes[i].innovationNo});
        child.genes[i].enabled = enabledGenes[i];
    }
    return child ;
}

/**
 * Gets the gene from a genome who's innovation number matches a given number
 * @param  {Genome} g    Genome to search in
 * @param  {Number} inno Innovation number to search from
 * @return {Number}      Index of the matching gene, or -1 if no gene matches
 */
function matchingGene(g, inno) {
    for (let i in g.genes) {
        if (g.genes[i].innovationNo == inno) {
            return i ;
        }
    }
    return -1 ;
}

/**
 * Mutates a genome
 * @param  {Genome} g Genome to mutate
 */
function mutate(g) {
    if (Math.random() < 0.8) {
        for(let i in g.genes) {
            g.genes[i].weight = mutateWeight(g.genes[i].weight);
        }
    }

    if (Math.random() < 0.05) {
        addConnection(g);
    }

    if (Math.random() < 0.03) {
        addNode(g);
    }
}

/**
 * Adds a node to a genome
 * @param {Genome} g Genome to mutate
 */
function addNode(g) {
    let randomConnection = 0; //The loop will assign the real value

    let availableConnection = false;
    for(let i in g.genes) {
        if(g.genes[i].from != g.biasNode) {
            availableConnection = true;
            break;
        }
    }

    if(availableConnection) {
        do {
            randomConnection = Math.floor(Math.random() * (g.genes.length));
        } while(g.genes[randomConnection].from == g.biasNode); //Do not disconnect
bias!

        g.genes[randomConnection].enabled = false;

        let newNodeNo = g.nextNode;
        g.nextNode++;

        let connectionInnovationNumber = getInnovationNumber(g,
g.genes[randomConnection].from, newNodeNo);
        g.genes.push({from : g.genes[randomConnection].from, to : newNodeNo, weight :
1, innovationNo : connectionInnovationNumber, enabled : true});


        connectionInnovationNumber = getInnovationNumber(g, newNodeNo,
g.genes[randomConnection].to);

        g.genes.push({from : newNodeNo, to : g.genes[randomConnection].to, weight :
g.genes[randomConnection].weight, innovationNo : connectionInnovationNumber, enabled :
true});
        g.nodes.push({no : newNodeNo, layer : getNode(g,
g.genes[randomConnection].from).layer + 1});

        connectionInnovationNumber = getInnovationNumber(g, g.biasNode, newNodeNo);

        g.genes.push({from : g.biasNode, to : newNodeNo, weight : 0, innovationNo :
connectionInnovationNumber, enabled : true});

        if(getNode(g, newNodeNo).layer == getNode(g,
g.genes[randomConnection].to).layer) {
```

```
            for (let i in g.nodes) {
                if (g.nodes[i].no != newNodeNo && g.nodes[i].layer >= getNode(g,
newNodeNo).layer) {
                    g.nodes[i].layer++;
                }
            }
            g.layers++;
        }
    }
}

/**
 * Gets the node whos id matches a given number
 * @param  {Genome} g  Genome to search in
 * @param  {Number} id Id to search for
 * @return {Node}    The matching node
 */
function getNode(g, id) {
    for(let i in g.nodes) {
        if(g.nodes[i].no == id) {
            return g.nodes[i];
        }
    }
}

/**
 * Adds a connection in a genome
 * @param {Genome} g Genome to mutate
 */
function addConnection(g) {
    if (fullyConnected(g)) {
        //Cannot add a connection to a full network
        return;
    }
    let randomNode1 = Math.floor(Math.random() * (g.nodes.length));
    let randomNode2 = Math.floor(Math.random() * (g.nodes.length));
    while (g.nodes[randomNode1].layer == g.nodes[randomNode2].layer ||
nodesConnected(g, g.nodes[randomNode1], g.nodes[randomNode2])) {
        randomNode1 = Math.floor(Math.random() * (g.nodes.length));
        randomNode2 = Math.floor(Math.random() * (g.nodes.length));
    }
    if (g.nodes[randomNode1].layer > g.nodes[randomNode2].layer) {
        let temp = randomNode2;
        randomNode2 = randomNode1;
        randomNode1 = temp;
    }
    let connectionInnovationNumber = getInnovationNumber(g, g.nodes[randomNode1].no,
g.nodes[randomNode2].no);
    g.genes.push({from : g.nodes[randomNode1].no, to : g.nodes[randomNode2].no,
weight : Math.random() * 2 - 1, innovationNo : connectionInnovationNumber, enabled :
true});
}

/**
 * Check if two given nodes are connected within a genome
 * @param  {Genome} g Genome to look into
 * @param  {Node}   a First node
 * @param  {Node}   b Second node
 * @return {Boolean}  True if the nodes are connected in any direction, false
otherwise
 */
function nodesConnected(g, a, b) {
    for(let i in g.genes) {
        if(g.genes[i].from == a.no && g.genes[i].to == b.no) {
            return true;
        } else if(g.genes[i].from == b.no && g.genes[i].to == a.no) {
            return true;
        }
    }
    return false;
}

/**
 * Check whether a genome is fully connected or not
 * @param  {Genome} g Genome to check
 * @return {Boolean}  True if the genome is fully connected, false otherwise
 */
function fullyConnected(g) {
    let maxConnections = 0;
    let nodesInLayers = Array.apply(null,
Array(g.layers)).map(Number.prototype.valueOf, 0);

    for (let i in g.nodes) {
        nodesInLayers[g.nodes[i].layer] += 1;
    }

    for (let i = 0; i < g.layers - 1; i++) {
        let nodesInFront = 0;
        for (let j = i + 1; j < g.layers; j++) {
            nodesInFront += nodesInLayers[j];
        }
        maxConnections += nodesInLayers[i] * nodesInFront ;
    }
    return maxConnections == g.genes.length ;
}

/**
 * Mutate a weight and gives the new value
 * @param  {Number} w Current weight
 * @return {Number}   The new, mutated, weight
 */
function mutateWeight(w) {
    if(Math.random() < 0.1) {
        return Math.random() * 2 - 1 ;
    } else {
        let neww = w + ((Math.random() - 0.5) / 20);
        if(neww > 1) {
            neww = 1 ;
        } else if(neww < -1){
            neww = -1 ;
        }
        return neww ;
    }
}

/**
 * Gets the innovation number for a connection (gene),
 * Or creates a new one if it's the first time it appears
 * @param  {Genome} g    Genome to search in
 * @param  {Number} from Node the gene starts from
 * @param  {Number} to   Node the gene ends to
 * @return {Number}      Innovation Number of the gene
 */
function getInnovationNumber(g, from, to) {
    let isNew = true ;
    let connectionInnovationNumber = nextConnectionNo ;
    for(let i in innovationHistory) {
        if(innovationMatches(g, innovationHistory[i], from, to)) {
            isNew = false ;
            connectionInnovationNumber = innovationHistory[i].innovationNumber ;
        }
    }

    if(isNew) {
        let innoNumbers = [];
        for(let i in g.genes) {
            innoNumbers.push(g.genes[i].innovationNo);
        }
        innovationHistory.push({from : from, to : to, innovationNumber :
connectionInnovationNumber, innovationNumbers : innoNumbers});
        nextConnectionNo++ ;
    }
    return connectionInnovationNumber ;
}

/**
 * Checks if the genome's genes are part of the innovation history
 * @param  {Genome} g               Genome to check
 * @param  {Innovation} innovation Innovation to look into
 * @param  {Number} from            Id of the input node
 * @param  {Number} to              Id of the output node
 * @return {Boolean}                True if all the genes are part of the innovation,
false otherwise
 */
function innovationMatches(g, innovation, from, to) {
    if (g.genes.length == innovation.innovationNumbers.length) {
        if (from == innovation.from && to == innovation.to) {
            for (let i in g.genes) {
                if (!innovation.innovationNumbers.includes(g.genes[i].innovationNo))
{
                    return false ;
                }
            }
            return true ;
        }
    }
    return false ;
}

const excessCoeff = 1.5 ;
const weightDiffCoeff = 0.8 ;
const compatibilityThreshold = 1 ;

function Specie(genome) {
    this.genomes = [genome];
    this.bestFitness = genome.fitness ;
    this.best = genome ;
    this.staleness = -1 ;
    this.averageFitness = 0 ;
}

/**
 * Checks if a genome is part of this specie
 * @param  {Genome} genome Genome to verify
 * @return {Boolean}       True if the genome is part of the specie, false otherwise
 */
Specie.prototype.sameSpecies = function(genome) {
    let excessAndDisjoint = this.getExcessDisjoint(genome, this.best);
    let averageWeightDiff = this.averageWeightDiff(genome, this.best);

    let compatibility = (excessCoeff * excessAndDisjoint) + (weightDiffCoeff *
averageWeightDiff);
    return (compatibilityThreshold > compatibility);
}

/**
 * Gets the excess and disjoint genes count between two genomes
 * @param  {Genome} g1 Genome 1
 * @param  {Genome} g2 Genome 2
 * @return {Number}    Amount of excess genes + amount of disjoint genes
 */
Specie.prototype.getExcessDisjoint = function(g1, g2) {
    let matching = 0;
    for (let i in g1.genes) {
        for (let j in g2.genes) {
            if(g1.genes[i].innovationNo == g2.genes[j].innovationNo) {
                matching++;
                break;
            }
        }
    }
    return (g1.genes.length + g2.genes.length - (2 * matching));
}

/**
 * Gets the average weight difference between every matching genes of two genomes
 * @param  {Genome} g1 Genome 1
 * @param  {Genome} g2 Genome 2
 * @return {Number}    Average weight difference
 */
Specie.prototype.averageWeightDiff = function(g1, g2) {
    let matching = 0.0;
    let totalDiff = 0.0;
    for(let i in g1.genes) {
        for(let j in g2.genes) {
            if(g1.genes[i].innovationNo == g2.genes[j].innovationNo) {
                matching++;
                totalDiff += Math.abs(g1.genes[i].weight - g2.genes[j].weight);
                break;
            }
        }
    }
    return (matching == 0 ? 100  : (totalDiff / matching));
}

/**
 * Adds a genome to the specie
 * @param  {Genome} genome Genome to add
 */
Specie.prototype.addToSpecies = function(genome) {
    this.genomes.push(genome);
    if(genome.fitness > this.bestFitness) {
        this.bestFitness = genome.fitness ;
        this.best = genome ;
        this.staleness = -1 ; //The staleness is going to be incremented back to 0
anyways
    }
}

/**
 * Calculate the average weight of the specie
 */
Specie.prototype.setAverage = function() {
    this.averageFitness = this.genomes.map(x => x.fitness).reduce((a, c) => a + c) /
this.genomes.length ;
}

/**
 * Kills the bottom half of the specie
 */
Specie.prototype.cull = function() {
    if(this.genomes.length > 2) {
        this.genomes.splice(Math.floor(this.genomes.length / 2), this.genomes.length -
Math.floor(this.genomes.length / 2));
    }
}

/**
 * Shares fitness between all genomes
 */
Specie.prototype.fitnessSharing = function() {
    for (let i in this.genomes) {
        this.genomes[i].fitness /= this.genomes.length ;
    }
}
```

```
/**
 * Removes all genome from the specie
 */
Specie.prototype.clear = function() {
    this.genomes = [];
}

/**
 * Creates an offspring from the specie
 * @return {Genome} Child made with genomes from the specie
 */
Specie.prototype.yieldChild = function() {
    let child = {};
    if (Math.random() < 0.25) {
        child = clone(this.select());
    } else {
        let p1 = clone(this.select());
        let p2 = clone(this.select());
        if (p1.fitness < p2.fitness) {
            child = crossover(p2, p1);
        } else {
            child = crossover(p1, p2);
        }
    }
    mutate(child);
    return child;
}

/**
 * Selects randomly a genome by taking in consideration it's fitness (CDF)
 * @param  {Array} population Whole population of genomes
 * @return {Genome}            The selected genome
 */
Specie.prototype.select = function() {
    let fitsum = this.genomes.map(x => x.fitness).reduce((a, c) => a + c);
    let threshold = Math.random() * fitsum;
    let sum = 0;
    for(let i in this.genomes) {
        sum += this.genomes[i].fitness;
        if(sum >= threshold) {
            return this.genomes[i];
        }
    }
    return this.genomes[0];
}
```

```
/**
 * Clones the given object
 *
 * @param  {Object} obj The object to clone
 * @return {Object} A clone of the given object
 */
function clone(obj) {
    return JSON.parse(JSON.stringify(obj));
}

/**
 * Sets the innovation history array
 * @param {Array} innohist New innovation history
 */
function setInnovationHistory(innohist) {
    innovationHistory = innohist;
}

/**
 * Sets the species array
 * @param {Array} _species New species array
 */
function setSpecies(_species) {
    species = _species;
}

/**
 * Gets the species array
 * @return {Array} Species
 */
function getSpecies() {
    return species;
}

/**
 * Gets the innovation history array
 * @return {Array} Innovation History
 */
function getInnovationHistory() {
    return innovationHistory;
}

module.exports.setInnovationHistory = setInnovationHistory;
module.exports.setSpecies = setSpecies;
module.exports.Specie = Specie;
module.exports.createNextGeneration = createNextGeneration ;
module.exports.getSpecies = getSpecies ;
module.exports.getInnovationHistory = getInnovationHistory ;
```

# 2 Client

Client.java

```java
/**
 * This files contains the program's entry point
 *
 * @author Thomas Ibanez
 */
package me.pv.mg.client ;

import me.pv.mg.client.genetic.GenomeCodec ;
import me.pv.mg.client.network.Network ;
import me.pv.mg.client.nn.NeuralNetwork ;
import me.pv.mg.client.simulation.AsteroidSimulator ;
import me.pv.mg.client.simulation.Simulator ;
import me.pv.mg.protobuf.Mg.MGNetworkType ;

/**
 * Program's main class, the class will be instantiated one time per thread
 */
public class Client extends Thread {

  private Network network ;
  private GenomeCodec gc ;
  private NeuralNetwork nn ;
  private Simulator sim ;

  private String name ;
  private boolean display ;

  public Client(String serverIP, String name, boolean display) {
    this.network = new Network(serverIP, this);
    this.gc = new GenomeCodec();
    this.name = name ;
    this.display = display ;
  }

  @Override
  public void run() {
    this.network.joinPool(name, display);
    while(true) {
      this.network.waitNextMessage();
    }
  }

  /**
   * Starts to compute a simulation of the given network on the given game
   * @param game     Name of the game
   * @param genome     String representation of the genome
   * @param metadata   Metadata of the genome
   * @param type     Type of network to build
   */
  public void startSimulation(String game, String genome, String metadata,
MGNetworkType type) {
    this.nn = this.gc.toNeuralNetwork(genome, metadata, type);
    if(game.equals("Asteroid")) {
      if(!display) {
        this.network.sendResponse(true);
      }
      this.sim = new AsteroidSimulator();
      long startTime = System.currentTimeMillis();
      float simFitness = this.sim.simulate(this.nn, display);
      if(!display) {
        this.network.sendResult(simFitness, (int) (System.currentTimeMillis() -
startTime));
      }
    } else {
      this.network.sendResponse(false);
    }
  }

  public static void main(String[] args) {
    if(args.length < 3) {
      System.out.println("Usage: client.jar <server_ip> <#threads> <name> [-s]");
      System.exit(1);
    }
    boolean spec = false;
    if(args.length > 3) {
      if(args[3].equals("-s")) {
        spec = true;
      }
    }
    int threads = spec ? 1  : Integer.parseInt(args[1]);
    Client[] clients = new Client[threads];
    for (int i = 0; i < threads; i++) {
      Client c = new Client(args[0], args[2], spec);
      clients[i] = c;
      c.start();
    }

    for (int i = 0; i < clients.length; i++) {
      try {
        clients[i].join();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

GenomeCodec.java

```
/**
 * This files contains the genomes decoder, it will be used to convert genome strings
to neural networks
 *
 * @author Thomas Ibanez
 */
package me.pv.mg.client.genetic;

import me.pv.mg.client.nn.ActivationFunctions;
import me.pv.mg.client.nn.MultilayerPerceptron;
import me.pv.mg.client.nn.NEATNetwork;
import me.pv.mg.client.nn.NeuralNetwork;
import me.pv.mg.protobuf.Mg.MGNetworkType;

public class GenomeCodec {

  /**
   * Converts a genome string to a netural network object
   * @param genome    Genome string
   * @param metadata  Metadata of the network
   * @param type    Type of the network
   * @return      Neural network corresponding to the genome
   */
  public NeuralNetwork toNeuralNetwork(String genome, String metadata, MGNetworkType
type) {
    String[] meta = metadata.split(",");
    int inputCount = Integer.parseInt(meta[0]);
    int hLayerCount = Integer.parseInt(meta[1]);
    int[] hLayers = new int[hLayerCount];
    for (int i = 0; i < hLayers.length; i++) {
      hLayers[i] = Integer.parseInt(meta[2 + i]);
    }
    int outputCount = Integer.parseInt(meta[2 + Math.max(1, hLayerCount)]);

    if (type == MGNetworkType.MG_MULTILAYER_PERCEPTRON) {
      String[] genomeInf = genome.split(",");
      float[] weights = new float[genomeInf.length];
      for (int i = 0; i < weights.length; i++) {
        weights[i] = Float.parseFloat(genomeInf[i]);
      }
      MultilayerPerceptron mlp = new MultilayerPerceptron(inputCount, hLayerCount,
hLayers, outputCount, ActivationFunctions::Sigmoid);
      mlp.setAllWeight(weights);
      return mlp;
    } else if (type == MGNetworkType.MG_NEAT) {
      String[] infos = genome.split(",");
      int genesCount = Integer.parseInt(infos[0]);
      int nodeCount = Integer.parseInt(infos[1]);
      int bias = Integer.parseInt(infos[2]);
      int layers = Integer.parseInt(infos[3]);

      NEATNetwork nn = new NEATNetwork(inputCount, outputCount, bias, layers,
ActivationFunctions::Sigmoid);

      for (int i = 4 + 3 * genesCount; i < 4 + 3 * genesCount + 2 * nodeCount; i +=
2) {
        int no = Integer.parseInt(infos[i]);
        int layer = Integer.parseInt(infos[i + 1]);
        nn.addNode(nn.new Node(no, layer));
      }

      for (int i = 4; i < 4 + 3 * genesCount; i += 3) {
        int from = Integer.parseInt(infos[i]);
        int to = Integer.parseInt(infos[i + 1]);
        float w = Float.parseFloat(infos[i + 2]);
        nn.addConnection(nn.new Connection(from, to, w));
      }
      nn.connect();
      return nn;
    }
    return null;
  }

}
```

Network.java

```
/**
 * This files contains the network related code
 *
 * @author Thomas Ibanez
 */
package me.pv.mg.client.network;

import java.io.DataInputStream;
import java.io.EOFException;
import java.io.IOException;
import java.net.Socket;
import java.nio.ByteBuffer;

import com.google.protobuf.GeneratedMessageV3;

import me.pv.mg.client.Client;
import me.pv.mg.protobuf.Mg.MGComputeRequest;
import me.pv.mg.protobuf.Mg.MGComputeResponse;
import me.pv.mg.protobuf.Mg.MGComputeResult;
import me.pv.mg.protobuf.Mg.MGJoin;
import me.pv.mg.protobuf.Mg.MGJoinResponse;
import me.pv.mg.protobuf.Mg.MGMessages;

public class Network {

  private Socket sock;
  private static final int PORT = 4567;
  private Client parent;
  private DataInputStream input;

  public Network(String ip, Client parent) {
    try {
      this.sock = new Socket(ip, PORT);
      this.parent = parent;
      this.input = new DataInputStream(this.sock.getInputStream());
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  /**
   * Send a message to join the pool of workers
   * @param name  The name to give to the server
   * @param spec  True if joining as a spectator, false otherwise
   */
  public void joinPool(String name, boolean spec) {
    MGJoin msg = MGJoin.newBuilder().setPrettyName(name).setSpectator(spec).build();
    sendMessage(MGMessages.MG_JOIN, msg);
  }

  /**
   * Sends the resulting fitness of a simulation to the server
   * @param fitness  The fitness to send
   * @param time    The time it took to compute the simulation (ms)
   */
  public void sendResult(float fitness, int time) {
    MGComputeResult msg =
MGComputeResult.newBuilder().setFitness(fitness).setTime(time).build();
    sendMessage(MGMessages.MG_COMPUTE_RESULT, msg);
  }

  /**
   * Sends a response to a request from the server
   * @param cando    True if the client is able to do the simulation, false otherwise
   */
  public void sendResponse(boolean cando) {
    MGComputeResponse msg = MGComputeResponse.newBuilder().setCanDo(cando).build();
    sendMessage(MGMessages.MG_COMPUTE_RESPONSE, msg);
  }

  /**
   * Waits for a message to come
   */
  public void waitNextMessage() {
    try {
      byte[] in_type = new byte[1];
      input.readFully(in_type);

      byte[] in_size = new byte[4];
      input.readFully(in_size);
      ByteBuffer bb = ByteBuffer.wrap(in_size);
      int size = bb.getInt();

      byte[] in_msg = new byte[size];
      input.readFully(in_msg);

      switch (MGMessages.forNumber(in_type[0])) {
        case MG_COMPUTE_REQUEST :
          MGComputeRequest cr = MGComputeRequest.parseFrom(in_msg);
          this.parent.startSimulation(cr.getComputeInfo().getGame(), cr.getGenome(),
cr.getComputeInfo().getNetMetadata(), cr.getComputeInfo().getNetType());
          break;
        case MG_END :
          System.exit(0);
          return;
        case MG_JOIN_RESPONSE :
          MGJoinResponse jr = MGJoinResponse.parseFrom(in_msg);
          if(jr.getAccepted() == false) {
            System.out.println("Join denied: "+jr.getReason());
            System.exit(0);
          }
          break;
        default :
          break ;
      }
    } catch(EOFException e) {
      //No Do
    } catch (IOException e) {
      e.printStackTrace();
      System.exit(1);
    }
  }

  /**
   * Sends a message to the server
   * @param type  Type of the message
   * @param msg  Message object
   */
  private void sendMessage(MGMessages type, GeneratedMessageV3 msg) {
    byte[] out = new byte[msg.getSerializedSize() + 2];
    out[0] = (byte) type.getNumber();
    out[1] = (byte) msg.getSerializedSize();
    for (int i = 0 ; i < msg.getSerializedSize(); i++) {
      out[2 + i] = msg.toByteArray()[i];
    }
    try {
      sock.getOutputStream().write(out);
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

}
```

ActivationFunction.java

```
/**
 * This files contains the interface for activation functions
 *
 * @author Thomas Ibanez
 */
package me.pv.mg.client.nn ;

public interface ActivationFunction {

  /**
   * Gets the activation result of the function
   * @param in    input to feed the function with
   * @return    The output of the function
   */
```

```java
  float activate(float in);

}
```

ActivationFunctions.java

```java
/**
 * This files contains some activations functions ready to be used
 *
 * @author Thomas Ibanez
 */
package me.pv.mg.client.nn;

public final class ActivationFunctions {

  public static float Sigmoid(float in) {
    return (float) (1.0 / (1.0 + Math.exp(-3 * in)));
  }

  public static float Sng(float in) {
    return in <= 0 ? 0  : 1;
  }

  public static float Tanh(float in) {
    return (float) Math.tanh(in);
  }

}
```

MultilayerPerceptron.java

```java
/**
 * This files contains the fully connected multilayer perceptron code
 *
 * @author Thomas Ibanez
 */
package me.pv.mg.client.nn;

import java.awt.Color;
import java.awt.Graphics;

public class MultilayerPerceptron extends NeuralNetwork {

  private float[] inToHid;
  private float[][] hidden;
   private float[] hidToOut;

   private float[][] hidValue;

   private int[] hLayers;

  public MultilayerPerceptron(int inputCount, int hLayerCount, int[] hLayers, int
outputCount, ActivationFunction activationFunction) {
    super(inputCount, outputCount, activationFunction);
    this.inToHid = new float[(inputCount + 1) * hLayers[0]];
    this.hidden = new float[hLayerCount - 1][];
    for (int i = 1; i < hLayers.length; i++) {
      this.hidden[i - 1] = new float[hLayers[i - 1] * hLayers[i]];
    }
    this.hidToOut = new float[hLayers[hLayerCount-1] * outputCount];

    this.hidValue = new float[hLayerCount][];
    for (int i = 0; i < hLayers.length; i++) {
      this.hidValue[i] = new float[hLayers[i]];
    }
    this.hLayers = hLayers;
  }

  /**
   * Sets all the weight of the network
   * @param weights  Array of the new weights
   */
  public void setAllWeight(float[] weights) {
    int offset = 0;
    for(int i = 0; i < this.inToHid.length; i++) {
      this.inToHid[i] = weights[i];
    }

    offset = this.inToHid.length;
    for (int i = 0; i < hidden.length; i++) {
      for (int j = 0; j < hidden[i].length; j++) {
        this.hidden[i][j] = weights[offset];
        offset++;
      }
    }
    for (int i = 0; i < hidToOut.length; i++) {
      this.hidToOut[i] = weights[offset + i];
    }
  }

  @Override
  public float[] propagateForward(float[] finput) {
    float[] input = new float[finput.length + 1];
    for (int i = 0; i < finput.length; i++) {
      input[i] = finput[i];
    }
    input[input.length - 1] = 1 ;

    for (int i = 0 ; i < this.hidValue.length ; i++) {
      for (int j = 0 ; j < this.hidValue[i].length ; j++) {
        float sum = 0 ;
        int lim = (i == 0 ? input.length  : this.hidValue[i - 1].length);
        for (int k = 0 ; k < lim ; k++) {
          if(i == 0) {
            sum += input[k] * inToHid[k + j * lim];
          } else {
            sum += this.hidValue[i-1][k] * hidden[i-1][k + j * lim];
          }
        }
        hidValue[i][j] = this.activationFunction.activate(sum);
      }
    }

    float[] output = new float[outputCount];
    for (int i = 0 ; i < outputCount ; i++) {
      float sum = 0 ;
      for (int j = 0 ; j < hidValue[hidValue.length - 1].length ; j++) {
        //take the furtest layer and forward to output
        sum += hidValue[hidValue.length - 1][j] * hidToOut[j + i *
hidValue[hidValue.length - 1].length];
      }
      output[i] = this.activationFunction.activate(sum);
    }
    return output ;
  }

  @Override
  public void display(Graphics g, int x, int y, int w, int h) {
    for(int i = 0 ; i <= hLayers.length + 1 ; i++) {
      if(i == 0) {
        for(int j = 0 ; j < inputCount + 1 ; j++) {
          int x1 = x + 5 ;
          int y1 = y + (j * h / (inputCount + 1));
          for(int k = 0 ; k < hLayers[0]; k++) {
            int x2 = x + 5 + (w / (hLayers.length + 2));
            int y2 = y + (k * h / hLayers[0]);
            if(inToHid[j + k * inputCount] > 0) {
              g.setColor(Color.GREEN);
            } else {
              g.setColor(Color.RED);
            }
      g.drawLine(x1 + 5, y1 + 5, x2 + 5, y2 + 5);
     }
     g.setColor(Color.BLACK);
     g.fillOval(x1, y1, 10, 10);
    }
   } else if(i == hLayers.length + 1) {
    for(int j = 0; j < outputCount; j++) {
        int x1 = x + 5 + (i * w / (hLayers.length + 2));
        int y1 = y + (j * h / outputCount);
        g.setColor(Color.BLACK);
        g.fillOval(x1, y1, 10, 10);
    }
   } else {
    for(int j = 0; j < hLayers[i - 1]; j++) {
     int x1 = x + 5 + (i * w / (hLayers.length + 2));
     int y1 = y + (j * h / hLayers[i - 1]);
     int lim = (i < hLayers.length ? hLayers[i]  : outputCount);
     for(int k = 0; k < lim; k++) {
      int x2 = x + 5 + ((i + 1) * w / (hLayers.length + 2));
      int y2 = y + (k * h /  lim);
      if((i < hLayers.length ? hidden[i - 1][j + k * lim]  : hidToOut[j + k *
lim]) > 0) {
       g.setColor(Color.GREEN);
      } else {
       g.setColor(Color.RED);
      }
      g.drawLine(x1 + 5, y1 + 5, x2 + 5, y2 + 5);
     }
     g.setColor(Color.BLACK);
     g.fillOval(x1, y1, 10, 10);
    }
   }
  }
 }
}
```

NEATNetwork.java

```
/**
 * This files contains the NEAT network (dynamic topology) code
 *
 * @author Thomas Ibanez
 */
package me.pv.mg.client.nn;

import java.awt.Color;
import java.awt.Graphics;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;

public class NEATNetwork extends NeuralNetwork {

  private List<Connection> genes;
  private Map<Integer, Node> nodes;
  private int bias, layers;

  public NEATNetwork(int inputCount, int outputCount, int bias, int layers,
ActivationFunction activationFunction) {
    super(inputCount, outputCount, activationFunction);
    this.genes = new ArrayList<>();
    this.nodes = new HashMap<>();
    this.bias = bias;
    this.layers = layers;
  }

  @Override
  public float[] propagateForward(float[] input) {
    for (Entry<Integer, Node> e : nodes.entrySet()) {
      e.getValue().setValue(0);
    }

    for (int i = 0; i < input.length; i++) {
      nodes.get(i).setValue(input[i]);
    }
    nodes.get(bias).setValue(1);

    for (int i = 0; i < layers; i++) {
      for (Entry<Integer, Node> e : nodes.entrySet()) {
        if (e.getValue().getLayer() == i) {
          if (i != 0) {
            e.getValue().setValue(activationFunction.activate(e.getValue().getValue()));
          }
          for (Connection c : e.getValue().getOutputs()) {
            nodes.get(c.getTo()).value += e.getValue().getValue() * c.getWeight();
          }
        }
      }
    }

    float[] outputs = new float[this.outputCount];
    for (int i = 0; i < outputs.length; i++) {
      outputs[i] = nodes.get(i + inputCount).value ;
    }
    return outputs ;
  }

  @Override
  public void display(Graphics g, int x, int y, int w, int h) {
    g.setColor(Color.black);
    int[] positions = new int[nodes.entrySet().size() * 2];
    int[] counts = new int[layers];
    int maxCount = 0 ;
    for (int i = 0 ; i < layers ; i++) {
      int c = 0 ;
      for (Entry<Integer, Node> e  : nodes.entrySet()) {
        if (e.getValue().getLayer() == i) {
          c++;
        }
      }
      if (c > maxCount)
        maxCount = c ;
      counts[i] = c ;
    }

    for (int i = 0 ; i < layers ; i++) {
      int j = 0 ;
      for (Entry<Integer, Node> e  : nodes.entrySet()) {
        if (e.getValue().getLayer() == i) {
          int dx = x + (((w - 20) / (layers - 1)) * i);
          int dy = y + ((h - 20) / maxCount) * j + ((maxCount - counts[i]) * (((h -
20) / 2) / maxCount)) + 5 ;
          g.fillOval(dx, dy, 10, 10);
          positions[e.getValue().no * 2] = dx + 5 ;
          positions[e.getValue().no * 2 + 1] = dy + 5 ;
          j++;
        }
      }
    }

    for (Connection connection  : genes) {
      if(connection.weight < 0) {
        g.setColor(Color.red);
      } else {
        g.setColor(Color.green);
      }
      if(connection.weight != 0) {
        g.drawLine(positions[connection.from * 2], positions[connection.from * 2 + 1],
positions[connection.to * 2], positions[connection.to * 2 + 1]);
      }
    }
  }

  /**
   * Connects the node from the genes informations, this function has to be called
before using the network
   */
  public void connect() {
    for (Connection connection  : genes) {
      this.nodes.get(connection.getFrom()).getOutputs().add(connection);
    }
  }

  /**
   * Adds a node to the network
   * @param n    The node to add
   */
  public void addNode(Node n) {
    this.nodes.put(n.no, n);
  }

  /**
   * Adds a connection to the network
   * @param c    The connection to add
   */
  public void addConnection(Connection c) {
    this.genes.add(c);
  }

  public class Node {
    private int no, layer;
    private final List<Connection> outputs;
    private float value;

    public Node(int no, int layer) {
      this.no = no;
      this.layer = layer;
      this.outputs = new ArrayList<>();
      this.setValue(0);
    }

    public int getNo() {
      return no;
  }

  public void setNo(int no) {
    this.no = no;
  }

  public int getLayer() {
    return layer;
  }

  public void setLayer(int layer) {
    this.layer = layer;
  }

  public List<Connection> getOutputs() {
    return outputs;
  }

  public float getValue() {
    return value;
  }

  public void setValue(float value) {
    this.value = value;
  }
}

public class Connection {

  private int from, to;
  private float weight;

  public Connection(int from, int to, float weight) {
    this.from = from;
    this.to = to;
    this.weight = weight;
  }

  public int getFrom() {
    return from;
  }

  public void setFrom(int from) {
    this.from = from;
  }
```

```
    public int getTo() {
      return to ;
    }

    public void setTo(int to) {
      this.to = to ;
    }

    public float getWeight() {
      return weight ;
    }

    public void setWeight(float weight) {
      this.weight = weight ;
    }
  }

}
```

NeuralNetwork.java

```
/**
 * This files contains the abstraction for any neural network
 *
 * @author Thomas Ibanez
 */
package me.pv.mg.client.nn ;

import java.awt.Graphics ;

public abstract class NeuralNetwork {

  protected int inputCount, outputCount ;
  protected ActivationFunction activationFunction ;

  public NeuralNetwork(int inputCount, int outputCount, ActivationFunction
activationFunction) {
    this.inputCount = inputCount ;
    this.outputCount = outputCount ;
    this.activationFunction = activationFunction ;
  }

  /**
   * Propagates the input through the neural network all the way until the outputs are
set
   * @param input    The input signals to propagates
   * @return      The output result
   */
  public abstract float[] propagateForward(float[] input);

  /**
   * Displays the neural network on the specified rectangle
   * @param g      The graphics to access the frame
   * @param x      The rectangle's top left corner x coordinate
   * @param y      The rectangle's top left corner y coordinate
   * @param w      The width of the rectangle
   * @param h      The height of the rectangle
   */
  public abstract void display(Graphics g, int x, int y, int w, int h);

}
```

AsteroidSimulator.java

```
/**
 * This files contains asteroids' simulator code
 *
 * @author Thomas Ibanez
 */
package me.pv.mg.client.simulation;

import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.Polygon;
import java.awt.RenderingHints;
import java.util.ArrayList;
import java.util.List;

import me.pv.mg.client.nn.NeuralNetwork;

public class AsteroidSimulator implements Simulator {

  private List<Asteroid> asteroids;
  private Bullet[] bullets;
  private Ship ship;
  private NeuralNetwork nn;
  public static final int WIDTH = 1000;
  public static final int HEIGHT = 720;
  private boolean forward = false;

  public AsteroidSimulator() {
    this.asteroids = new ArrayList<>();
    this.ship = new Ship();
    this.bullets = new Bullet[5];
  }

  @Override
  public void paint(Graphics2D g) {
    g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);

    nn.display(g, WIDTH, 0, 300, HEIGHT);

    g.setColor(Color.black);
    g.fillRect(0, 0, WIDTH, HEIGHT);
    g.setColor(Color.WHITE);

    for (Asteroid asteroid  : new ArrayList<>(asteroids)) {
      int s = asteroid.size * Asteroid.RENDER_MULT ;
      g.drawOval((int) (asteroid.x - s / 2f), (int) (asteroid.y - s / 2f), s, s);
    }

    for (Bullet bullet  : bullets) {
      if (bullet  != null)
        g.drawOval((int) bullet.x, (int) bullet.y, 2, 2);
    }

    g.translate((int) ship.x, (int) ship.y);
    g.rotate(ship.angle);
    g.drawPolygon(ship.poly);
    if(forward) {
            g.drawPolygon(ship.boost);
    }
  }

  @Override
  public float simulate(NeuralNetwork nn, boolean display) {
    this.nn = nn ;
    float tick = 0 ;
    float score = 1 ;
    float hits = 1 ;
    float shots = 1 ;
    int ascount = 4 ;
    int spawnCountdown = 10000 ;
    int bulletTime = 60 ;
    Display frame = null ;
    if (display) {
      frame = new Display(WIDTH + 300, HEIGHT, this);
      frame.setVisible(true);
    }

    while (ship.isAlive()) {
      float[] input = new float[8];
      for (int i = 0 ; i < 8 ; i++) {
        double angle = 2 * Math.PI * i / 8 ;
        float vx = (float) (Math.cos(angle + ship.angle));
        float vy = (float) (Math.sin(angle + ship.angle));
        float min = Float.MAX_VALUE ;
        for (Asteroid asteroid  : new ArrayList<>(asteroids)) {
          for (int j = 0 ; j < 4 ; j++) {
            float ax = 0, ay = 0 ;
            if (j == 0) {
              ax = asteroid.x ;
              ay = asteroid.y ;
            } else if (j == 1) {
              ax = asteroid.x > WIDTH / 2 ? asteroid.x - WIDTH  : asteroid.x + WIDTH ;
              ay = asteroid.y > HEIGHT / 2 ? asteroid.y - HEIGHT  : asteroid.y +
HEIGHT ;
            } else if (j == 2) {
              ax = asteroid.x > WIDTH / 2 ? asteroid.x - WIDTH  : asteroid.x + WIDTH ;
              ay = asteroid.y ;
            } else {
              ax = asteroid.x ;
              ay = asteroid.y > HEIGHT / 2 ? asteroid.y - HEIGHT  : asteroid.y +
HEIGHT ;
            }
            float ux = ax - ship.x ;
            float uy = ay - ship.y ;
            float dot = ux * vx + uy * vy ;
            if (dot < 0) {
              continue ;
            }
            float normu = (float) Math.sqrt(ux * ux + uy * uy);
            float projx = dot * vx ;
            float projy = dot * vy ;
            float anglevu = (float) Math.acos(dot / normu);
            if (anglevu > Math.PI / 8) {
              continue ;
            }
            float distProj = (float) Math.sqrt(projx * projx + projy * projy);
            if (distProj - asteroid.size * Asteroid.RENDER_MULT / 2 < min) {
              min = distProj - asteroid.size * Asteroid.RENDER_MULT / 2 ;
            }
          }
        }
        input[i] = 1.0f / min ;
      }
```

```
    float[] out = nn.propagateForward(input);

    for (int i = 0; i < bullets.length; i++) {
      if (bullets[i] == null && bulletTime == 0 && out[0] > 0.8) {
        bulletTime = 60;
        shots++;
        bullets[i] = new Bullet((int) ship.x, (int) ship.y, ship.angle);
      } else if (bullets[i] != null) {
        if (bullets[i].ttl <= 0) {
          bullets[i] = null;
          continue;
        }
        bullets[i].x += bullets[i].vx;
        bullets[i].y += bullets[i].vy;
        if (bullets[i].x < 0) {
          bullets[i].x = WIDTH - 1;
        }
        if (bullets[i].y < 0) {
          bullets[i].y = HEIGHT - 1;
        }
        bullets[i].x %= WIDTH;
        bullets[i].y %= HEIGHT;
        bullets[i].ttl--;
      }
    }

    if (out[1] > 0.8 && out[1] > out[2]) {
      ship.angle += 0.08f;
    } else if (out[2] > 0.8 && out[2] > out[1]) {
      ship.angle -= 0.08f;
    }

    if (out[3] > 0.8) {
      ship.forward();
      forward = true;
    } else {
      forward = false;
    }
    ship.update();

    if (bulletTime > 0) {
      bulletTime--;
    }
    if (asteroids.size() == 0 || spawnCountdown-- == 0) {
      spawnCountdown = 10000;
```

```
        if(asteroids.size() == 0 && ascount  != 4) {
          score += 10 ;
        }
        for (int i = 0 ; i < ascount ; i++) {
          if (i == 0 && ascount == 4) {
            Asteroid a = new Asteroid();
            a.x = 0 ;
            a.y = 0 ;
            a.vy = (float) 1.5f ;
            a.vx = (float) 2 ;
            asteroids.add(a);
          } else
            asteroids.add(new Asteroid());
        }
        ascount++;
      }


      for (Asteroid asteroid  : new ArrayList<>(asteroids)) {
        float dist = (float) Math.sqrt((asteroid.x - ship.x) * (asteroid.x - ship.x) +
(asteroid.y - ship.y) * (asteroid.y - ship.y));
        if (dist < ((asteroid.size * Asteroid.RENDER_MULT / 2) + (ship.SIZE / 2))) {
          ship.setAlive(false);
        }

        for (int i = 0 ; i < bullets.length ; i++) {
          if (bullets[i]  != null) {
            dist = (float) Math.sqrt((asteroid.x - bullets[i].x) * (asteroid.x -
bullets[i].x) + (asteroid.y - bullets[i].y) * (asteroid.y - bullets[i].y));
            if (dist < asteroid.size * Asteroid.RENDER_MULT / 2) {
              score++;
              hits++;
              bullets[i] = null ;
              if (asteroid.size > 1) {
                for (Asteroid a  : asteroid.split()) {
                  asteroids.add(a);
                }
              }
              asteroids.remove(asteroid);
            }
          }
        }
        asteroid.x += asteroid.vx ;
        asteroid.y += asteroid.vy ;
        if (asteroid.x < 0) {
          asteroid.x = WIDTH - 1 ;
```

```
    }
    if (asteroid.y < 0) {
     asteroid.y = HEIGHT - 1;
    }
    asteroid.x %= WIDTH;
    asteroid.y %= HEIGHT;
   }

   if (display) {
    try {
     frame.setTitle("Asteroid | Fitness: "+(tick * score * 10 + score * (hits /
shots) * (hits / shots)));
     Thread.sleep(10);
    } catch (InterruptedException e) {
     e.printStackTrace();
    }
    frame.repaint();
   }
   tick++;
  }
  if (frame != null) {
   try {
    Thread.sleep(1000);
   } catch (InterruptedException e) {
    e.printStackTrace();
   }
   frame.dispose();
  }
  return tick * score * 10 + score * (hits / shots) * (hits / shots);
 }

 class Asteroid {
  private float x, y;
  private float vx, vy;
  private int size = 4;
  private static final int RENDER_MULT = 32;

  public Asteroid() {
   float dist = 0;
   do {
       double x = Math.random();
       double y = Math.random();
       this.x = (float) (x * WIDTH);
       this.y = (float) (y * HEIGHT);
       dist = (float) Math.sqrt((this.x - ship.x) * (this.x - ship.x) + (this.y
- ship.y) * (this.y - ship.y));
```

```
    } while(dist < this.size * Asteroid.RENDER_MULT * 2);
    this.vx = (float) (Math.random() * 4) - 2 ;
    this.vy = (float) (Math.random() * 4) - 2 ;
  }

  /**
   * Splits the asteroid into 2 little asteroids
   * @return    Array with 2 asteroid objects
   */
  public Asteroid[] split() {
    Asteroid[] childs = new Asteroid[2];
    for (int i = 0 ; i < childs.length ; i++) {
      childs[i] = new Asteroid();
      childs[i].x = this.x ;
      childs[i].y = this.y ;
      childs[i].vx = (float) (this.vx + (Math.random() / 2));
      childs[i].vy = (float) (this.vy + (Math.random() / 2));
      childs[i].size = this.size / 2 ;
    }
    return childs ;
  }
}

class Bullet {
  private float x, y ;
  private float vx, vy ;
  private float ttl = 110 ;

  public Bullet(int x, int y, float angle) {
    this.x = x ;
    this.y = y ;
    this.vx = (float) (Math.cos(angle) * 6);
    this.vy = (float) (Math.sin(angle) * 6);
  }
}

class Ship {
  private float x, y ;
  private float vx, vy ;
  private float angle ;
  private final int SIZE = 40 ;
  private final int MAX_SPEED = 10 ;
  private boolean alive = true ;
  private Polygon poly ;
  private Polygon boost ;
```

```
    public Ship() {
      poly = new Polygon(new int[] { SIZE / 2, -SIZE / 2, -SIZE / 2 }, new int[] { 0,
SIZE / 3, -SIZE / 3 }, 3);
      boost = new Polygon(new int[] { -SIZE, -SIZE / 2, -SIZE / 2 }, new int[] { 0,
SIZE / 6, -SIZE / 6 }, 3);
      this.x = WIDTH / 2;
      this.y = HEIGHT / 2;
      this.angle = (float) -(Math.PI / 2);
    }

    /**
     * Move the ship forward
     */
    public void forward() {
      this.vx += Math.cos(angle) * 0.3;
      this.vy += Math.sin(angle) * 0.3;
      if (this.vx * this.vx + this.vy * this.vy > MAX_SPEED * MAX_SPEED) {
        float div = (float) Math.sqrt(this.vx * this.vx + this.vy * this.vy);
        this.vx = this.vx / div * MAX_SPEED;
        this.vy = this.vy / div * MAX_SPEED;
      }
    }

    /**
     * Update the ship position and speed
     */
    public void update() {
      this.x += vx;
      this.y += vy;
      this.vx /= 1.02f;
      this.vy /= 1.02f;
      if (this.x < 0) {
        this.x = WIDTH - 1;
      }
      if (this.y < 0) {
        this.y = HEIGHT - 1;
      }
      this.x %= WIDTH;
      this.y %= HEIGHT;
    }

    public float getAngle() {
      return angle;
    }

    public void setAngle(float angle) {
      this.angle = angle;
    }

    public float getX() {
      return x;
    }

    public void setX(float x) {
      this.x = x;
    }

    public float getY() {
      return y;
    }

    public void setY(float y) {
      this.y = y;
    }

    public boolean isAlive() {
      return alive;
    }

    public void setAlive(boolean alive) {
      this.alive = alive;
    }
  }
}
```

Display.java

```
/**
 * This files contains a jframe class to display game progress
 *
 * @author Thomas Ibanez
 */
package me.pv.mg.client.simulation;

import java.awt.BorderLayout;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;
```

```
public class Display extends JFrame implements KeyListener {

  private static final long serialVersionUID = 7155015806747010932L ;
  private DisplayPanel contentPane ;
  private Simulator parent ;

  public boolean up, down, left, right ;

  public Display(int width, int height, Simulator s) {
    setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    setBounds(100, 100, width, height);
    contentPane = new DisplayPanel();
    contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
    contentPane.setLayout(new BorderLayout(0, 0));
    setContentPane(contentPane);
    addKeyListener(this);
    this.parent = s ;
  }

  class DisplayPanel extends JPanel {
    private static final long serialVersionUID = -8357439850680313124L ;

    @Override
    public void paint(Graphics g) {
      parent.paint((Graphics2D) g.create());
    }
  }

  @Override
  public void keyTyped(KeyEvent e) {

  }

  @Override
  public void keyPressed(KeyEvent e) {
    if(e.getKeyCode()== KeyEvent.VK_RIGHT)
      right = true ;
        else if(e.getKeyCode()== KeyEvent.VK_LEFT)
            left = true ;
        else if(e.getKeyCode()== KeyEvent.VK_DOWN)
            down = true ;
        else if(e.getKeyCode()== KeyEvent.VK_UP)
            up = true ;
  }

  @Override
  public void keyReleased(KeyEvent e) {
    if(e.getKeyCode()== KeyEvent.VK_RIGHT)
      right = false;
        else if(e.getKeyCode()== KeyEvent.VK_LEFT)
            left = false;
        else if(e.getKeyCode()== KeyEvent.VK_DOWN)
            down = false;
        else if(e.getKeyCode()== KeyEvent.VK_UP)
            up = false;
  }

}
```

Simulator.java

```
/**
 * This files contains the simulator interface
 *
 * @author Thomas Ibanez
 */
package me.pv.mg.client.simulation;

import java.awt.Graphics2D;

import me.pv.mg.client.nn.NeuralNetwork;

public interface Simulator {

  /**
   * Simulates the whole game and computes the fitness
   * @param nn      Neural network to simuate with
   * @param display  True if the game should be displayed, false otherwise
   * @return      Fitness of the network
   */
  float simulate(NeuralNetwork nn, boolean display);

  /**
   * Draws the simulation
   * @param g    Graphics to draw with
   */
  void paint(Graphics2D g);
}
```

# 3 Protocole

mg.proto

```
syntax = "proto2" ;
option java_package = "me.pv.mg.protobuf" ;

enum MGMessages {
    MG_JOIN = 1 ;
    MG_JOIN_RESPONSE = 2 ;
    MG_COMPUTE_REQUEST = 3 ;
    MG_COMPUTE_RESPONSE = 4 ;
    MG_COMPUTE_RESULT = 5 ;
    MG_END = 6 ;
}

enum MGNetworkType {
    MG_MULTILAYER_PERCEPTRON = 1 ;
    MG_NEAT = 2 ;
}

message MGJoin {
    optional string pretty_name = 1 ;
    optional bool spectator = 2 ;
}

message MGJoinResponse {
    required bool accepted = 1 ;
    optional string reason = 2 ;
}

message MGComputeInfo {
    required string game = 1 ;
    required .MGNetworkType net_type = 3 ;
    required string net_metadata = 4 ;
}

message MGComputeRequest {
    required .MGComputeInfo compute_info = 1 ;
    required string genome = 2 ;
}

message MGComputeResponse {
    required bool can_do = 1 ;
}

message MGComputeResult {
    required float fitness = 1 ;
    optional uint32 time = 2;
}

message MGEnd {
    optional string message = 1;
}
```