Agent zero tutorial

Table of contents

Section 1: Starting with Agent Zero

Everyone have to start somewhere!

In the following section we will cover the basics of installing agent zero and running your algorithms.

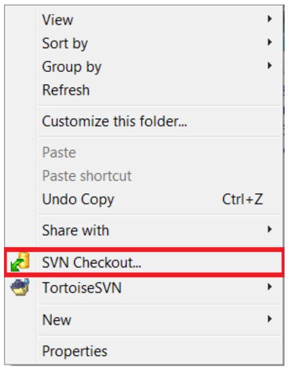
If you are already familiar with agent zero skip this section – you may want to start at Section 2 for more advanced topics.

Chapter 1: Downloading and setting up the eclipse plug-in

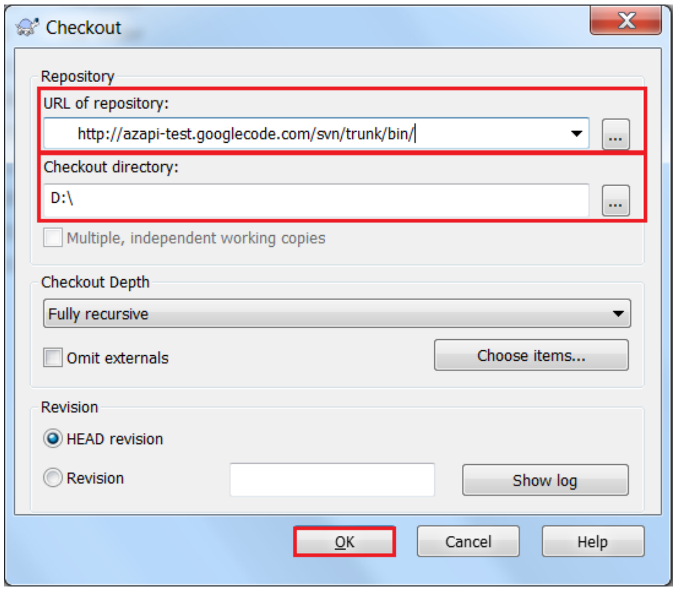
For the download you need to use SVN client. This demonstration is using Tortoise SVN, but you can use any client you know.

Step 1: downloading the plug-in:

a. Download and install Tortoise SVN from here.
b. Create a new folder named agentzero.
c. Left click on the agentzero folder.
d. In the menu that opens choose SVN checkout.



e. In the checkout screen enter on the "URL of repository" field the following address: http://azapi-test.googlecode.com/svn/trunk/bin/ and press OK.

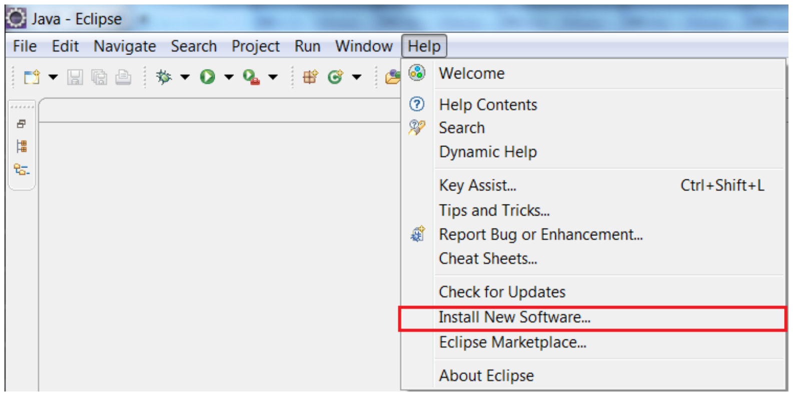


URL of repository: http://azapi-test.googlecode.com/svn/trunk/bin/
Checkout directory: D:\

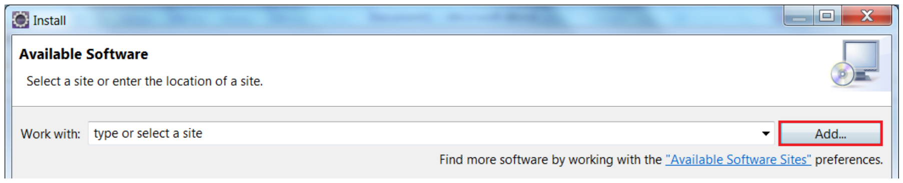
f. Wait until the download is complete.

Step 2: installing the plug-in:

a. Open eclipse and go to help=> install new software.



b. On the “Install” screen press “Add”.



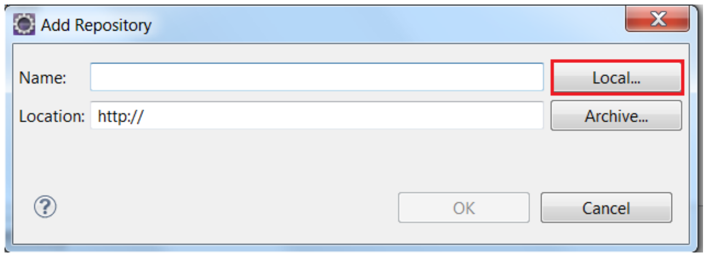
Available Software

Select a site or enter the location of a site.

Work with: type or select a site

Find more software by working with the "Available Software Sites" preferences.

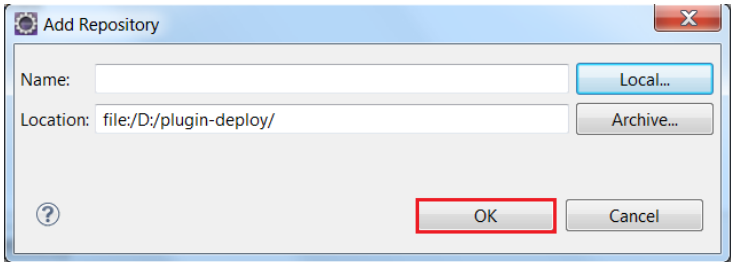
c. On the “Add Repository” screen press “local”, choose the agentzero folder and press “OK”.



Name:

Location: http://

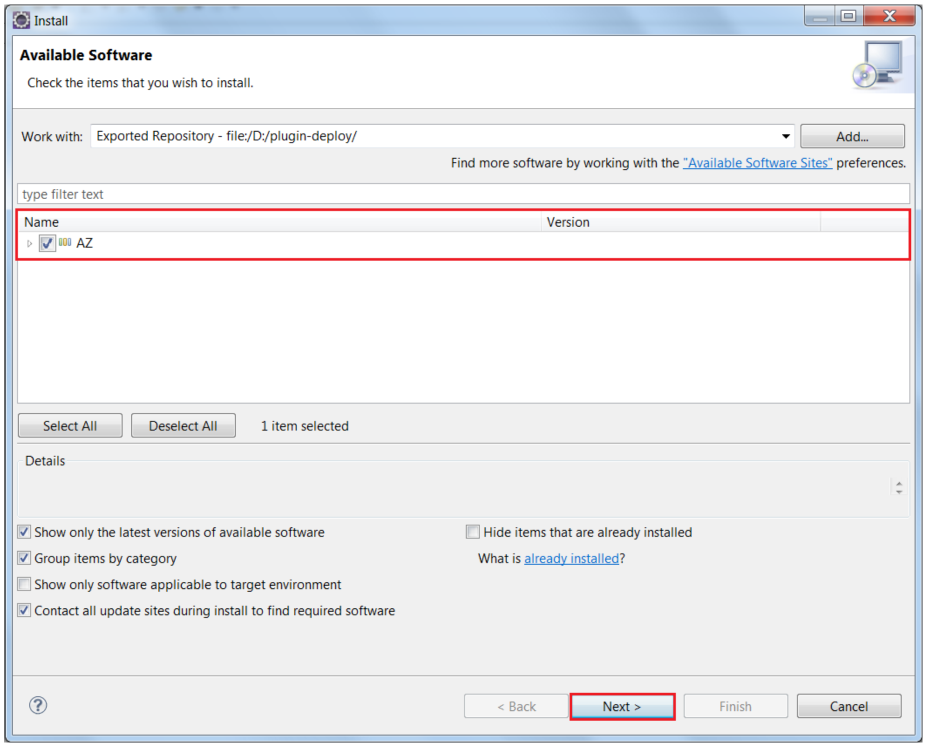
d. Enter the name “AgentZero” in the name field and press “OK”.



Name:

Location: file:/D:/plugin-deploy/

e. Check the category AZ and press “Next”.



Install

Available Software

Check the items that you wish to install.

Work with: Exported Repository - file:/D:/plugin-deploy/

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
✓ AZ	

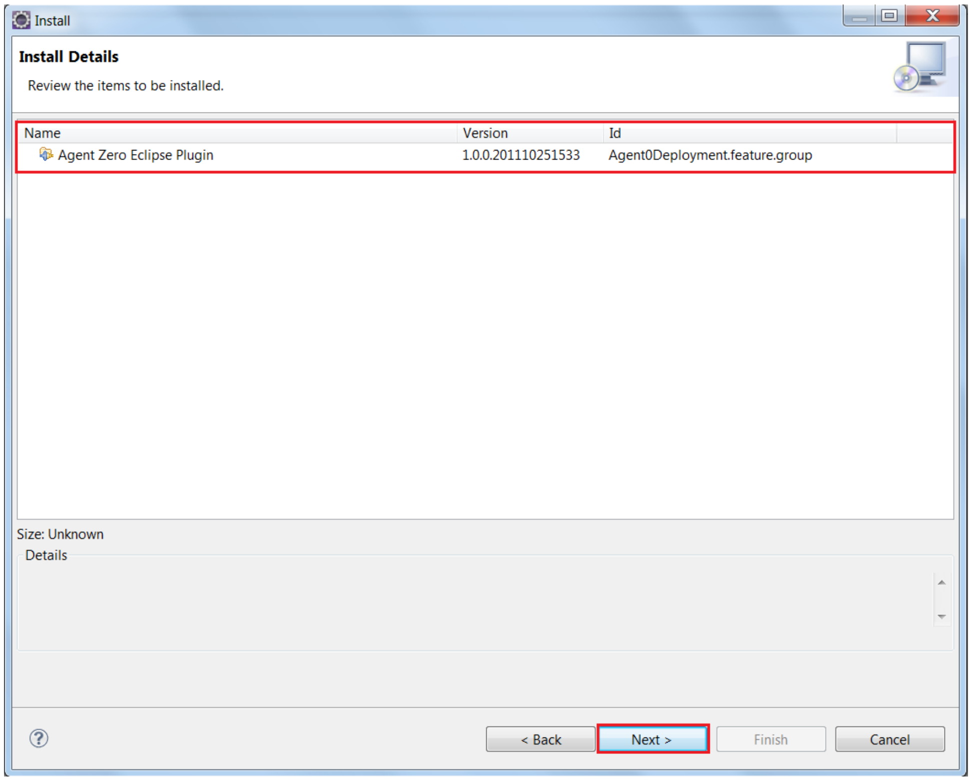
1 item selected

Details

- ✓ Show only the latest versions of available software
- ✓ Group items by category
- ☐ Show only software applicable to target environment
- ✓ Contact all update sites during install to find required software
- ☐ Hide items that are already installed

What is already installed?

f. Press “Next” again to begin the installation.



Install

Install Details

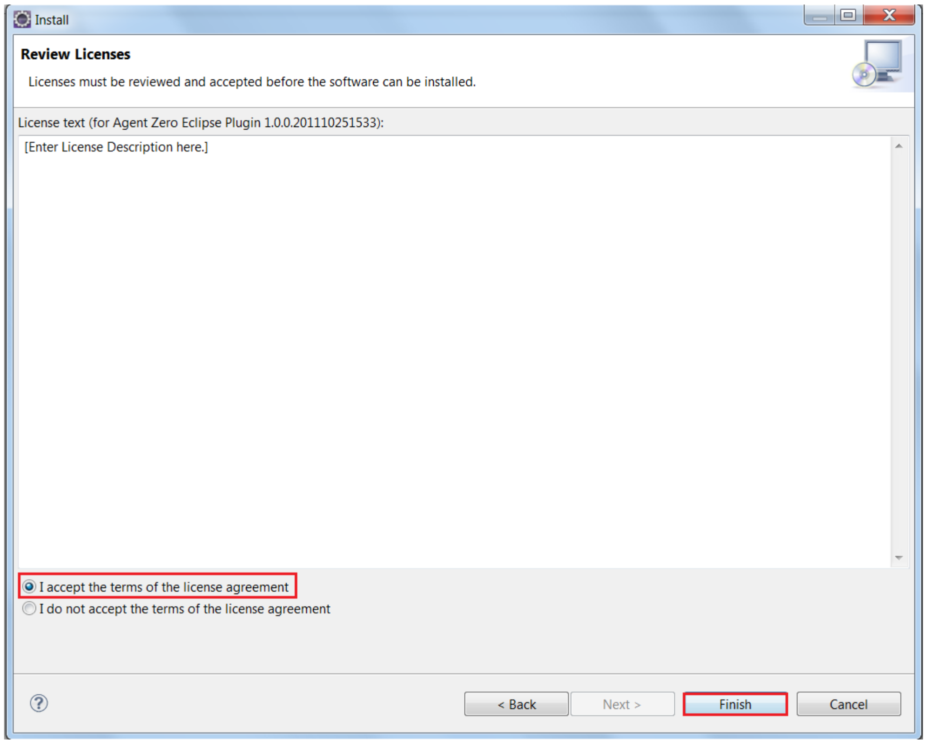
Review the items to be installed.

Name	Version	Id
Agent Zero Eclipse Plugin	1.0.0.201110251533	Agent0Deployment.feature.group

Size: Unknown

Details

g. Check "I accept the terms of the license agreement" and press "Finish".


Install

Review Licenses

Licenses must be reviewed and accepted before the software can be installed.

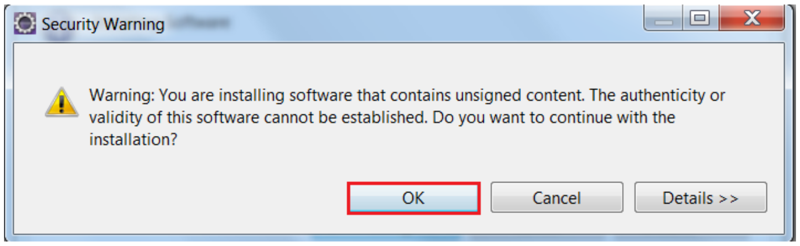
License text (for Agent Zero Eclipse Plugin 1.0.0.201110251533):

[Enter License Description here.]

[✓ I accept the terms of the license agreement]

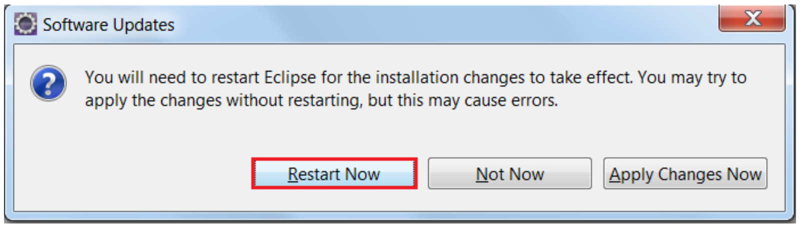
I do not accept the terms of the license agreement

h. If a "Security Warning" screen will appear, just press "OK".


Security Warning

Warning: You are installing software that contains unsigned content. The authenticity or validity of this software cannot be established. Do you want to continue with the installation?

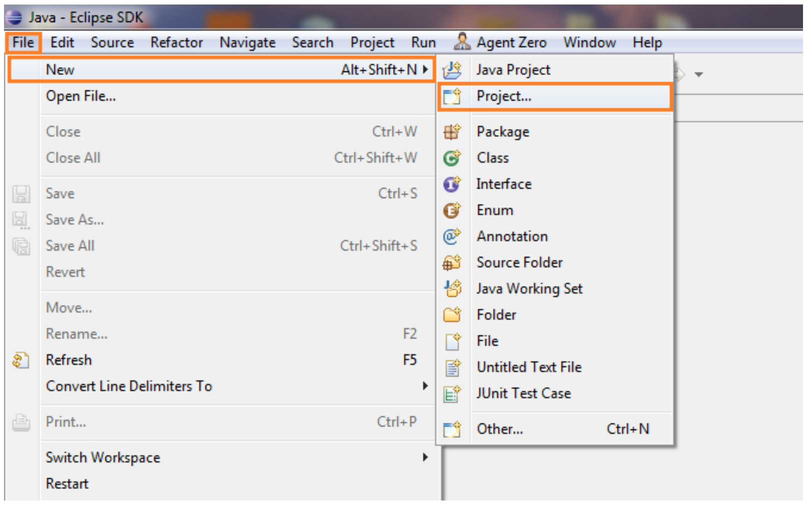
i. In the "Software Updates" screen press "Restart Now".


Software Updates

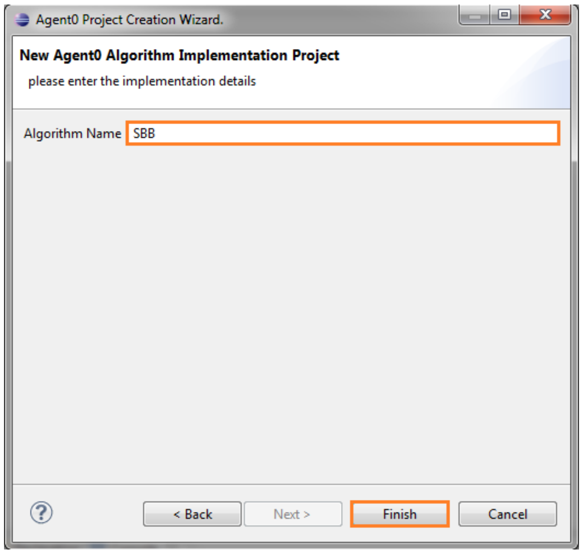
You will need to restart Eclipse for the installation changes to take effect. You may try to apply the changes without restarting, but this may cause errors.

Chapter 2: Creating your first AgentZero project

a. Open Eclipse and press File=> New=> Project.

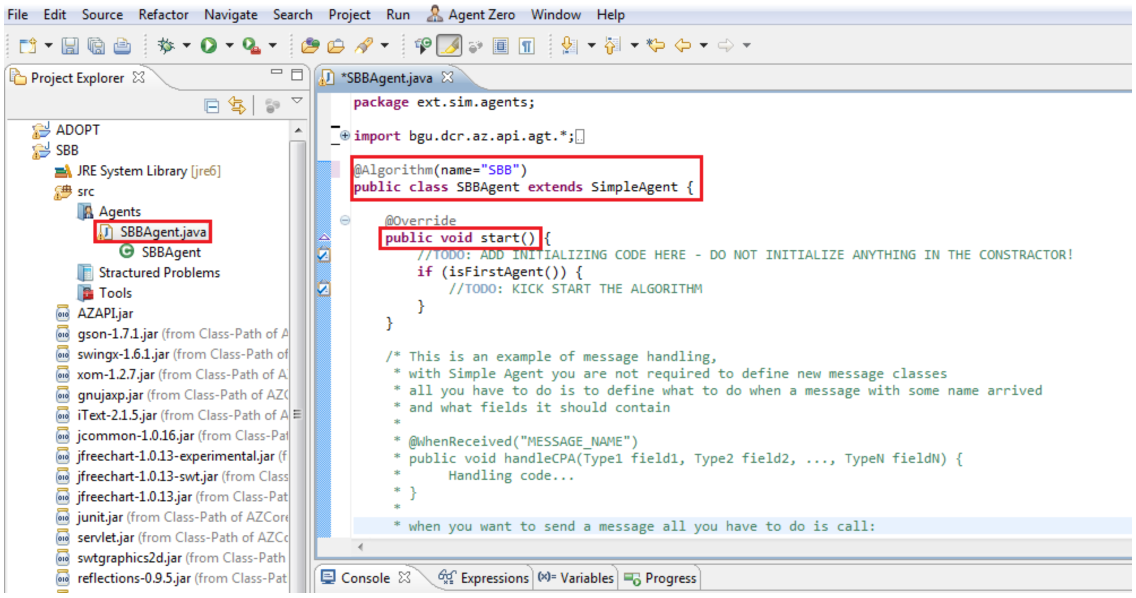


b. In the “Agent0 Project Creation wizard” fill the ‘Algorithm Name” field as you wish and press “Finish”. For the purpose of this tutorial, it will be called SBB.


Agent0 Project Creation Wizard.
New Agent0 Algorithm Implementation Project
please enter the implementation details
< Back
Next >
Finish
Cancel

Algorithm Name: SBB

c. Navigate to src=> ext.sim.agents=> SBBAgent in the package explorer.
You will see that a minimal template has been created for you.


File Edit Source Refactor Navigate Search Project Run Agent Zero Window Help
Project Explorer
*SBBAgent.java
ADOPT
SBB
JRE System Library [jre6]
src
Agents
SBBAgent.java
SBBAgent
Structured Problems
Tools
AZAPI.jar
gson-1.7.1.jar (from Class-Path of A
swingx-1.6.1.jar (from Class-Path of
xom-1.2.7.jar (from Class-Path of A
gnujaxp.jar (from Class-Path of AZC
iText-2.1.5.jar (from Class-Path of A
jcommon-1.0.16.jar (from Class-Pat
jfreechart-1.0.13-experimental.jar (f
jfreechart-1.0.13-swt.jar (from Class
jfreechart-1.0.13.jar (from Class-Pat
junit.jar (from Class-Path of AZCore
servlet.jar (from Class-Path of AZCc
swtgraphics2d.jar (from Class-Path
reflections-0.9.5.jar (from Class-Pat
Console
Expressions
Variables
Progress

```
package ext.sim.agents;

import bgu.dcr.az.api.agt.*;

@Algorithm(name="SBB")
public class SBBAgent extends SimpleAgent {

    @Override
    public void start() {
        //TODO: ADD INITIALIZING CODE HERE - DO NOT INITIALIZE ANYTHING IN THE CONSTRACTOR!
        if (isFirstAgent()) {
            //TODO: KICK START THE ALGORITHM
        }
    }

    /* This is an example of message handling,
     * with Simple Agent you are not required to define new message classes
     * all you have to do is to define what to do when a message with some name arrived
     * and what fields it should contain
     *
     * @WhenReceived("MESSAGE_NAME")
     * public void handleCPA(Type1 field1, Type2 field2, ..., TypeN fieldN) {
     *     Handling code...
     * }
     *
     * when you want to send a message all you have to do is call:
```

Chapter 3: writing your first algorithm

For starters, we encourage you to read the JavaDoc and especially the Agent class. At this writing, the JavaDoc can be downloaded via SVN in the following link: http://azapi-test.googlecode.com/svn/trunk/doc/.
In this tutorial, we will cover the most common API parts, but you can check-out http://azapi-test.googlecode.com/svn/trunk/alg/ to see the already implemented algorithms.

Algorithm Entry Point:

Simple agent is an abstract class which requires the implementation of the function start().
This is the entry point of the algorithm. The function start() will be called once for each agent.

Defining Messages:

The message handling methodology in SimpleAgent is that messages are actually remote procedure calls. Defining a new message means defining a new method and annotating it with @WhenRecieved annotation (there is a shortcut for this in the plugin : Alt+M). The message arguments are the parameters of the method. The message also contains Metadata for passing data about the message (timestamp, sender, etc.).
The actual message object can be retrieved by calling currentMessage() for cases in which you need access to the message.

Sending Messages:

Message can be sent via one of the two methods:
* broadcast(MessageName, Args…) - will broadcast the message to all agent Except the sending agent.
* send(MessageName, Args…).to*(…) – will send the message to the agents corresponding to the type of "to".

The sent messages will be posted via the mail service, to the relevant agent message queue and when the time is right, the remote procedure will get called.

Hooks:

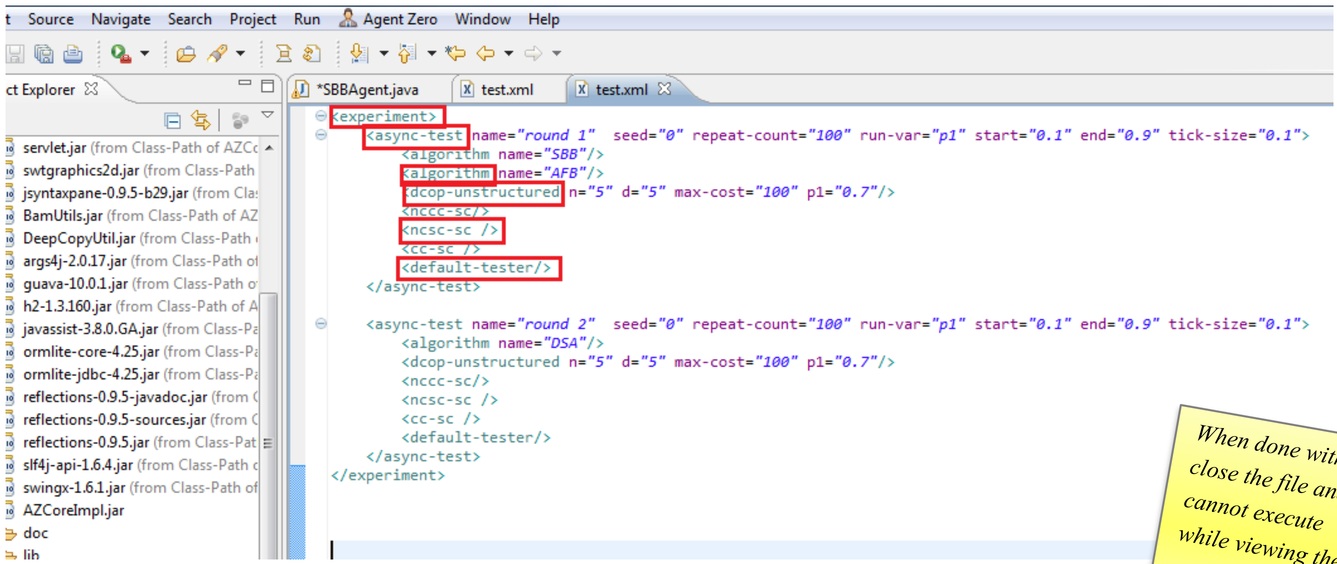
A hook is a method that is used to alter the behavior of the agent's basic functionality.
At this writing, there are two hooks available:
* beforeMessageSending(Message) - override this function in case you want to make some action every time before sending a message. This is a great place to write logs, attach timestamps to the message metadata etc.
* beforeMessageProcessing(Message) - override this method to perform preprocessing before messages arrive to their remote procedures. This is the place to change the message or even return completely other one. In the case of returning null the message is rejected and dumped.

Those are the very basic knowledge that is needed for implementing an algorithm. You should read the JavaDoc for more details.

Chapter 4: Running and debugging your algorithm

Running and debugging your algorithm in the Eclipse plug in is very simple.

Before running the algorithm, let’s take a look at the test.xml in the root of your project. At this writing, there is no suitable editor for editing the test.xml, but it is a simple xml and you should be able to edit it by hand.
let’s look at this file structure:

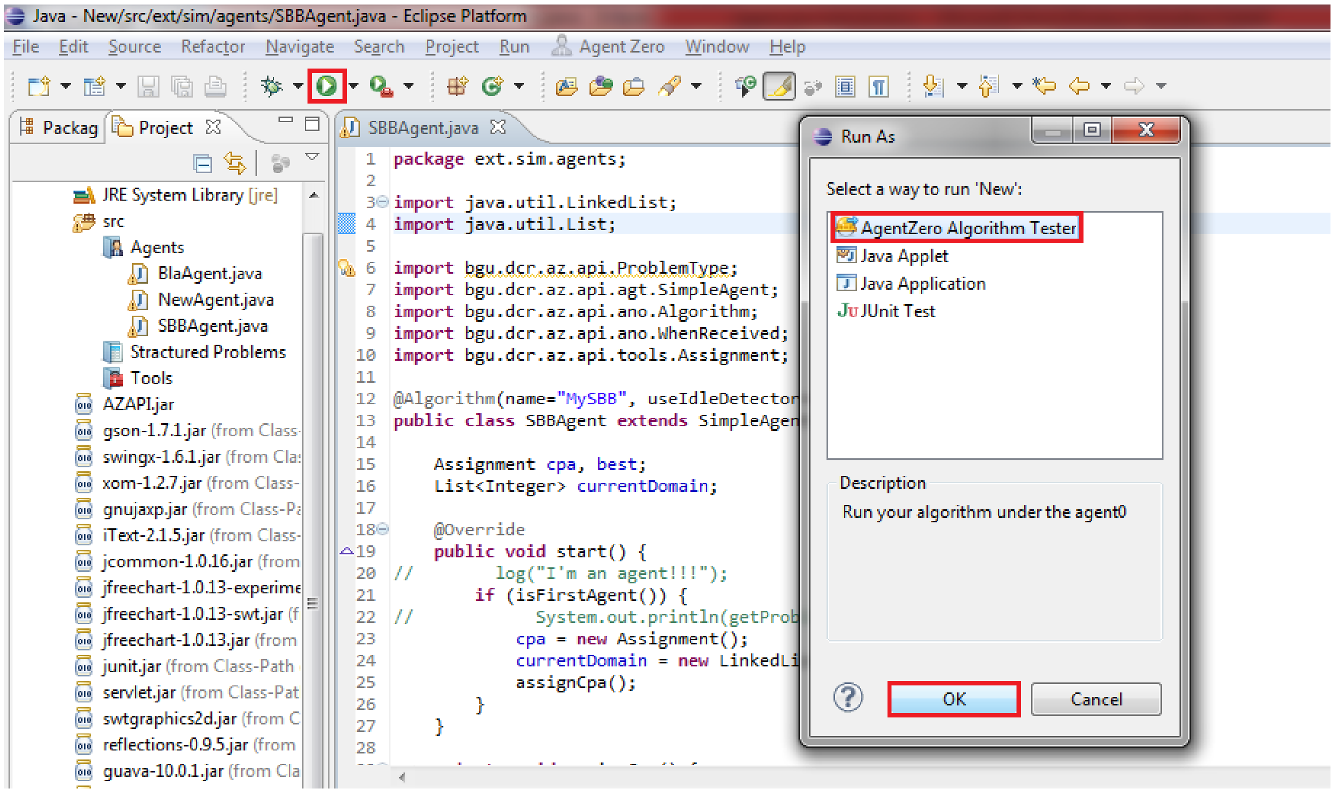

t Source Navigate Search Project Run Agent Zero Window Help
ct Explorer
*SBBAgent.java
test.xml
test.xml
servlet.jar (from Class-Path of AZC
swtgraphics2d.jar (from Class-Path
jsyntaxpane-0.9.5-b29.jar (from Cla
BamUtils.jar (from Class-Path of AZ
DeepCopyUtil.jar (from Class-Path
args4j-2.0.17.jar (from Class-Path o
guava-10.0.1.jar (from Class-Path o
h2-1.3.160.jar (from Class-Path of A
javassist-3.8.0.GA.jar (from Class-P
ormlite-core-4.25.jar (from Class-P
ormlite-jdbc-4.25.jar (from Class-P
reflections-0.9.5-javadoc.jar (from
reflections-0.9.5-sources.jar (from
reflections-0.9.5.jar (from Class-Pat
slf4j-api-1.6.4.jar (from Class-Path
swingx-1.6.1.jar (from Class-Path of
AZCoreImpl.jar
doc
lib

```
<experiment>
    <async-test name="round 1"  seed="0" repeat-count="100" run-var="p1" start="0.1" end="0.9" tick-size="0.1">
        <algorithm name="SBB"/>
        <algorithm name="AFB"/>
        <dcop-unstructured n="5" d="5" max-cost="100" p1="0.7"/>
        <nccc-sc/>
        <ncsc-sc />
        <cc-sc />
        <default-tester/>
    </async-test>

    <async-test name="round 2"  seed="0" repeat-count="100" run-var="p1" start="0.1" end="0.9" tick-size="0.1">
        <algorithm name="DSA"/>
        <dcop-unstructured n="5" d="5" max-cost="100" p1="0.7"/>
        <nccc-sc/>
        <ncsc-sc />
        <cc-sc />
        <default-tester/>
    </async-test>
</experiment>
```

When done with test.xml – close the file and save it. You cannot execute while viewing the file.

As we can see, the root node defines an experiment. Experiment is a collection of rounds. Each test represents

To execute your project press on the “Play” button, Choose “AgentZero Algorithm Tester”, and press “OK”.


Java - New/src/ext/sim/agents/SBBAgent.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Agent Zero Window Help
Packag
Project
JRE System Library [jre]
src
Agents
BlaAgent.java
NewAgent.java
SBBAgent.java
Structured Problems
Tools
AZAPI.jar
gson-1.7.1.jar (from Class-
swingx-1.6.1.jar (from Cla
xom-1.2.7.jar (from Class-
gnujaxp.jar (from Class-Pa
iText-2.1.5.jar (from Class-
jcommon-1.0.16.jar (from
jfreechart-1.0.13-experime
jfreechart-1.0.13-swt.jar (f
jfreechart-1.0.13.jar (from
junit.jar (from Class-Path
servlet.jar (from Class-Pat
swtgraphics2d.jar (from C
reflections-0.9.5.jar (from
guava-10.0.1.jar (from Cla
SBBAgent.java

```
package ext.sim.agents;

import java.util.LinkedList;
import java.util.List;

import bgu.dcr.az.api.ProblemType;
import bgu.dcr.az.api.agt.SimpleAgent;
import bgu.dcr.az.api.ano.Algorithm;
import bgu.dcr.az.api.ano.WhenReceived;
import bgu.dcr.az.api.tools.Assignment;

@Algorithm(name="MySBB", useIdleDetector
public class SBBAgent extends SimpleAgen

    Assignment cpa, best;
    List<Integer> currentDomain;

    @Override
    public void start() {
//          log("I'm an agent!!!");
        if (isFirstAgent()) {
//              System.out.println(getProb
            cpa = new Assignment();
            currentDomain = new LinkedLi
            assignCpa();
        }
    }
```

Run As

Select a way to run 'New':

- AgentZero Algorithm Tester
- Java Applet
- Java Application
- JUnit Test

Description

Run your algorithm under the agent0

OK Cancel

This will start your execution. Follow us to the next section to learn how to control running and debugging.

SECTION 2: RUNNING AND DEBUGGING

In this section we are going to learn about the running and debugging modes; how to use the development UI in order to preform simple result analyzing and helpful debug hints.

Your development cycle should be as follows:
first you should write an algorithm, than you should run an experiment.

If the algorithm fails during the experiment, the problem which caused the failure will be saved, and you should be able to run your algorithm in debug mode, and debug the saved problem.

So let's start by learning how to create an experiment.

Experiments are defined in a XML file, named test.xml, located in your project root folder.
The XML root is experiment. Within the experiment, you can add rounds. Currently, there are only two test types available: sync-test and async-test. As the naming implies, they perform sync and async execution (despicably). A test is a special type of loop. It can run any variable and perform algorithm execution in respect to that variable. Let's examine the test attributes:

name: each test should have a unique name for you to identify it when its running
seed: this is an optional field. Two rounds with the same seed and the same problem generators definition will produce the same problems.
run-var: within the test, there are collection of modules. Those modules have variables. A test can run one of those variables. For example, some problem generators define "p2" variable, so you can run "p2". in case you don't want to run any variable, just to loop the same configuration, you can assign this value to "_".
tick-size: for each value of run-var, the test will generate "tick-size" problems.
start: the start value of run-var
end: the end value of run-var
tick: the increment in the value of run-var

After you define the test, you should add algorithms for running in the test. You do so by adding a child "algorithm" to the test, the algorithm has one attribute-"name" which is the name that is written in the algorithm annotation within the agent class.

Next step, define the problem generator for the test. There are 4 basic random generators (and you can add your own – see problem generator chapter).
The basic generators are:
- unstructured-dscp
- unstructured-dcop
- unstructured-adcop
- connected-dcop

Each of these generators receives the following attributes:
n: number of variables
d: domain size
max-cost: the maximum cost of constraint
p1: probability of constraint between two variables
p2: probability of conflict between two constrained variables

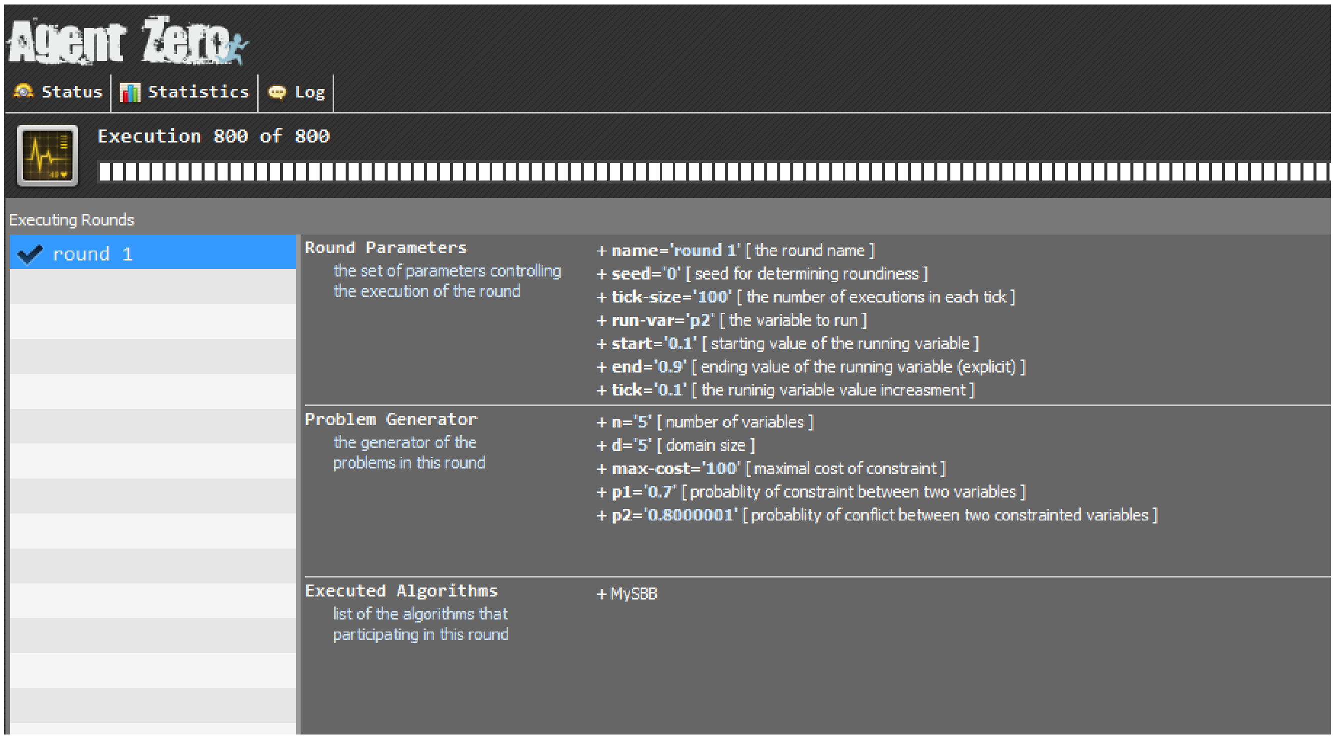
Optionally, you can add a correctness tester. Currently, there is one correctness tester defined:
default-tester – which receives no attributes and tests the solution of complete algorithms using centralized algorithms: branch & bound for dcop and mac-fc for dscp.

The last thing you can do is to add statistic analyzers. Currently, there are 3 analyzers:
-nccc-sc : number of concurrent constraint checks
-ncsc-sc : number of concurrent steps of computation
-cc-sc : number of constraint checks
all the analyzer attributes are controlled within the UI so you don’t have to supply them.

After you define your experiment, you can either run or debug it.

Chapter 1: Running

In Eclipse, press on the play button and choose "AgentZero Algorithm Tester"; the testing environment should start:



Execution 800 of 800

Executing Rounds

round 1

Round Parameters
the set of parameters controlling the execution of the round

+ **name**='round 1' [the round name]
+ **seed**='0' [seed for determining roundiness]
+ **tick-size**='100' [the number of executions in each tick]
+ **run-var**='p2' [the variable to run]
+ **start**='0.1' [starting value of the running variable]
+ **end**='0.9' [ending value of the running variable (explicit)]
+ **tick**='0.1' [the runinig variable value increasment]

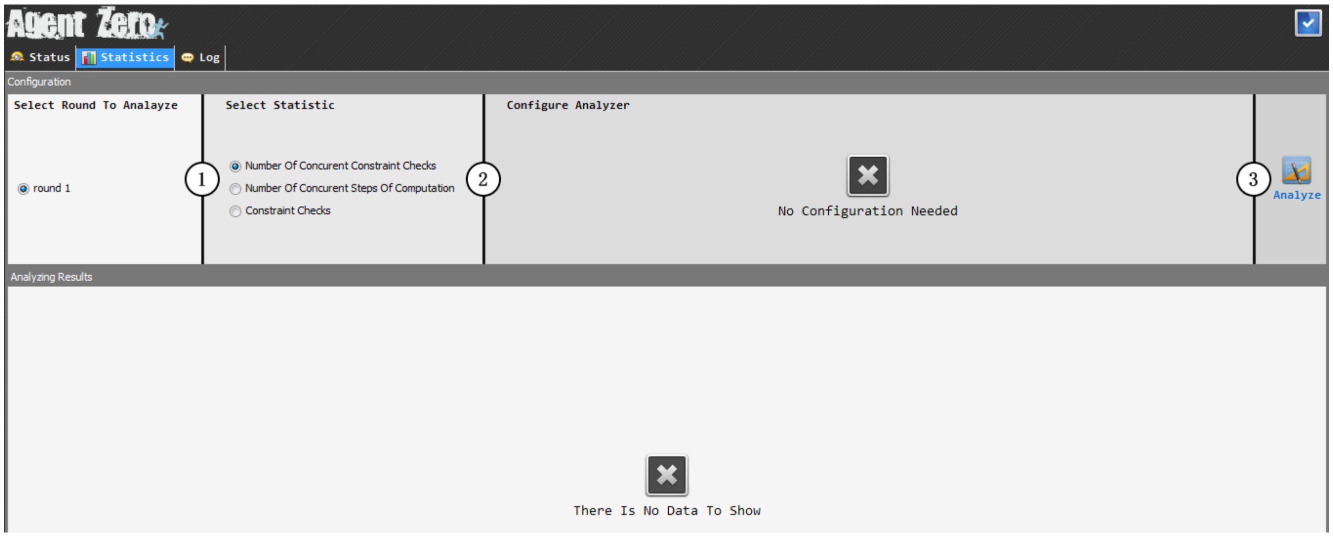
Problem Generator
the generator of the problems in this round

+ **n**='5' [number of variables]
+ **d**='5' [domain size]
+ **max-cost**='100' [maximal cost of constraint]
+ **p1**='0.7' [probablity of constraint between two variables]
+ **p2**='0.8000001' [probablity of conflict between two constrainted variables]

Executed Algorithms
list of the algorithms that participating in this round

+ MySBB

At the status screen, you can look at the definition of your rounds, and the progress of the execution.
After a test is complete, you can switch to the statistics screen to analyze statistics about it:



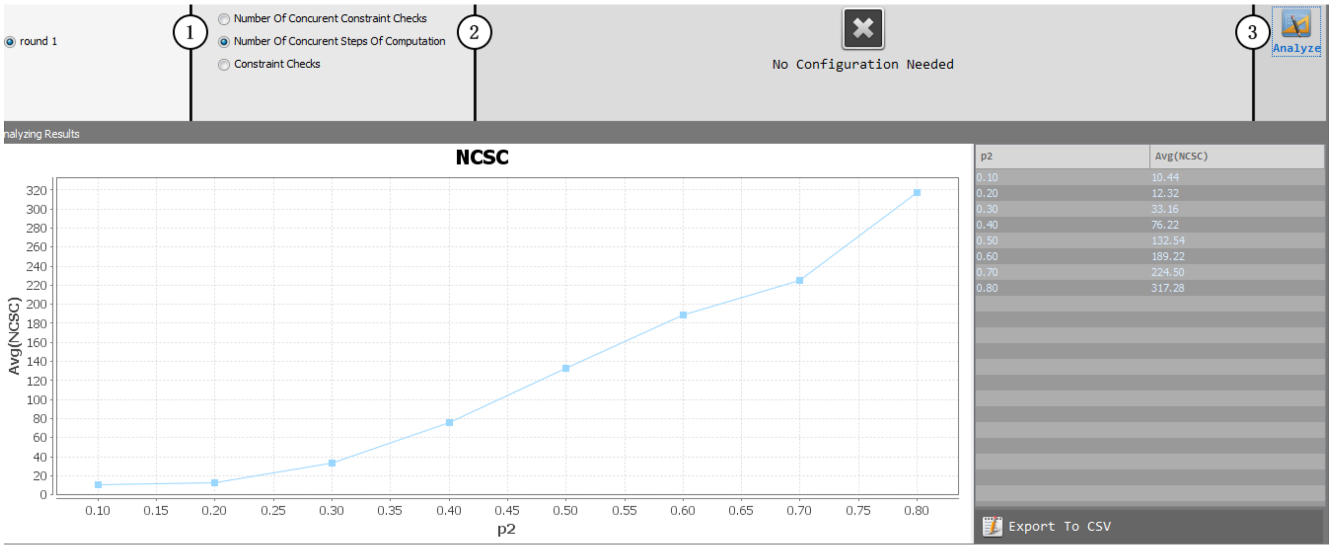
[✓ round 1]

[✓ Number Of Concurent Constraint Checks]
Number Of Concurent Steps Of Computation
Constraint Checks

No Configuration Needed

There Is No Data To Show

In this screen, you can select the test you want to analyze, and then you can select any of the statistics that were produced by the statistic analyzers that were added to the test in the experiment.
Next, if the analyzer requires any attribute, you can configure it and finally press “analyze”:



[✓ round 1]

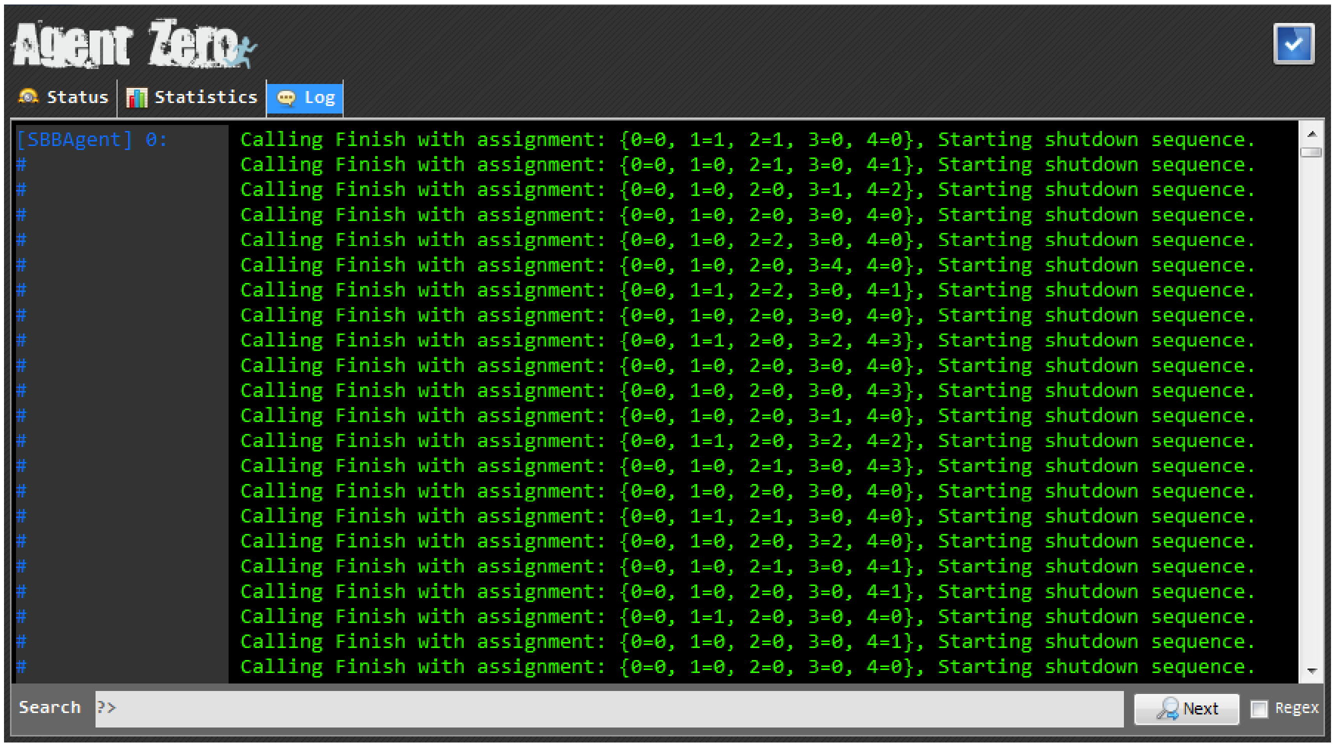
Number Of Concurent Constraint Checks
[✓ Number Of Concurent Steps Of Computation]
Constraint Checks

No Configuration Needed

p2	Avg(NCSC)
0.10	10.44
0.20	12.32
0.30	33.16
0.40	76.22
0.50	132.54
0.60	189.22
0.70	224.50
0.80	317.28

the analyzing is shown within the left graph and the raw data in the right table. You can export the raw data to CSV for more analyzing VIA Excel or any other spreadsheet program by pressing “Export To CSV” button.

While the algorithm is running, agents can produce logs. You can switch to the log screen anytime to view these logs:



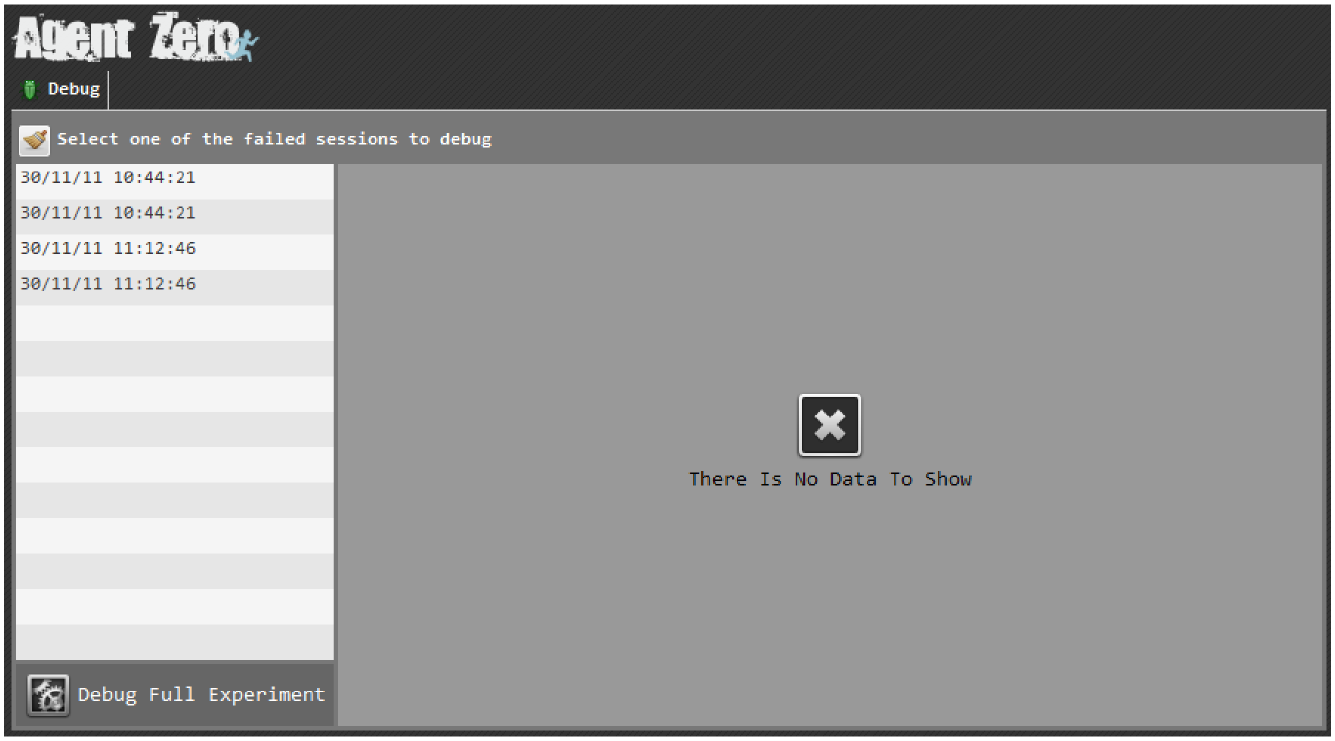
```
[SBBAgent] 0:    Calling Finish with assignment: {0=0, 1=1, 2=1, 3=0, 4=0}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=0, 2=1, 3=0, 4=1}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=0, 2=0, 3=1, 4=2}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=0, 2=0, 3=0, 4=0}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=0, 2=2, 3=0, 4=0}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=0, 2=0, 3=4, 4=0}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=1, 2=2, 3=0, 4=1}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=0, 2=0, 3=0, 4=0}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=1, 2=0, 3=2, 4=3}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=0, 2=0, 3=0, 4=0}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=0, 2=0, 3=0, 4=3}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=0, 2=0, 3=1, 4=0}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=1, 2=0, 3=2, 4=2}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=0, 2=1, 3=0, 4=3}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=0, 2=0, 3=0, 4=0}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=1, 2=1, 3=0, 4=0}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=0, 2=0, 3=2, 4=0}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=0, 2=1, 3=0, 4=1}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=0, 2=0, 3=0, 4=1}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=1, 2=0, 3=0, 4=0}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=0, 2=0, 3=0, 4=1}, Starting shutdown sequence.
#                Calling Finish with assignment: {0=0, 1=0, 2=0, 3=0, 4=0}, Starting shutdown sequence.
Search ?>
```

You can search in the logs using the lower bar – it supports textual search and regex search (by selecting the regex checkbox in the right).

Chapter 2: Debugging

While developing algorithms, it is not uncommon that an execution will fail (either by crushing or returning wrong solution while there is correctness tester defined). In this unfortunate case, AgentZero will save the problematic problem and by pressing debug in eclipse you can rerun it in debug mode.
In this mode, apart from all the eclipse debug functionality, there are several features which AgentZero provides for a nicer and easier debugging process.

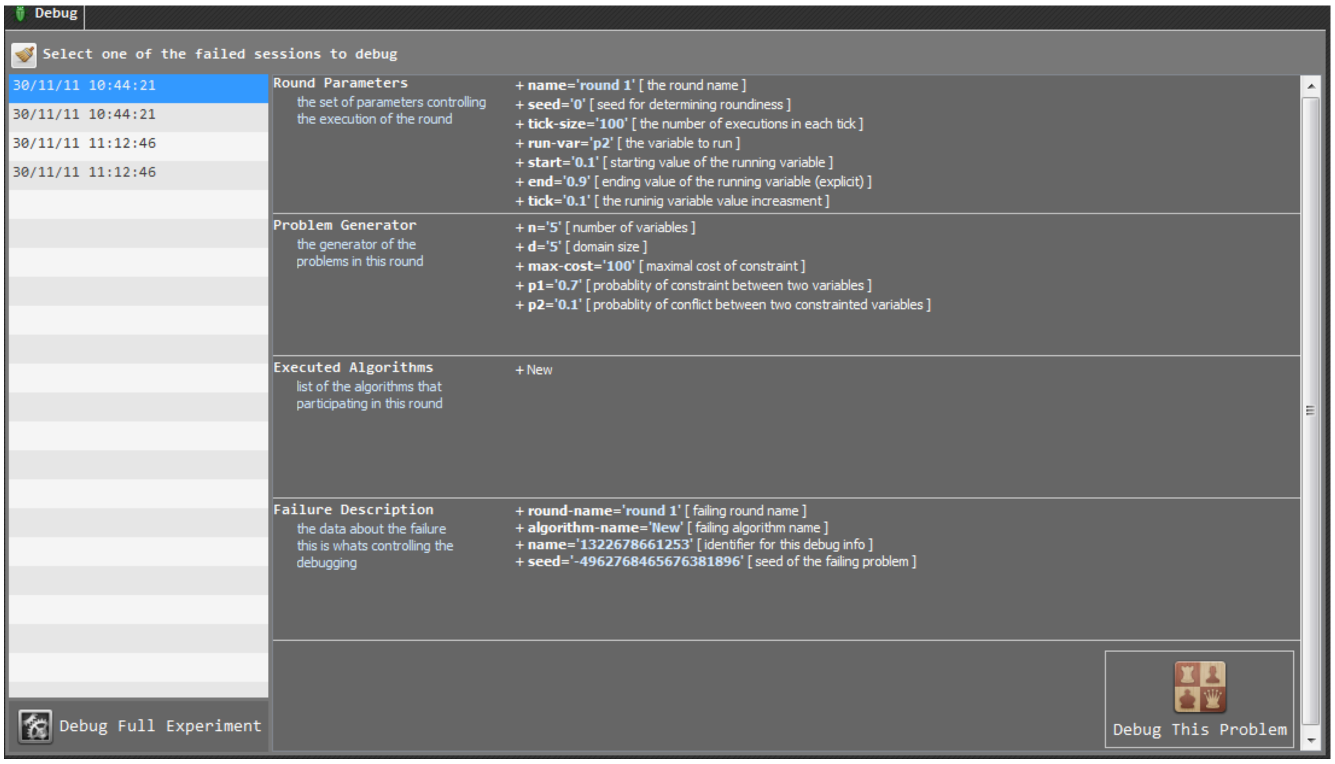
Let's start by pressing the debug button (we assume that you have run your algorithms and found a "bad" problem):



```
30/11/11 10:44:21
30/11/11 10:44:21
30/11/11 11:12:46
30/11/11 11:12:46
```

The debug screen will appear with a list of failed problems (you can also run your full experiment in debug mode by pressing the "Debug Full Experiment" on the bottom left corner).

Upon pressing one of the failed problems, the information about it will appear in the right panel:



30/11/11 10:44:21
30/11/11 10:44:21
30/11/11 11:12:46
30/11/11 11:12:46

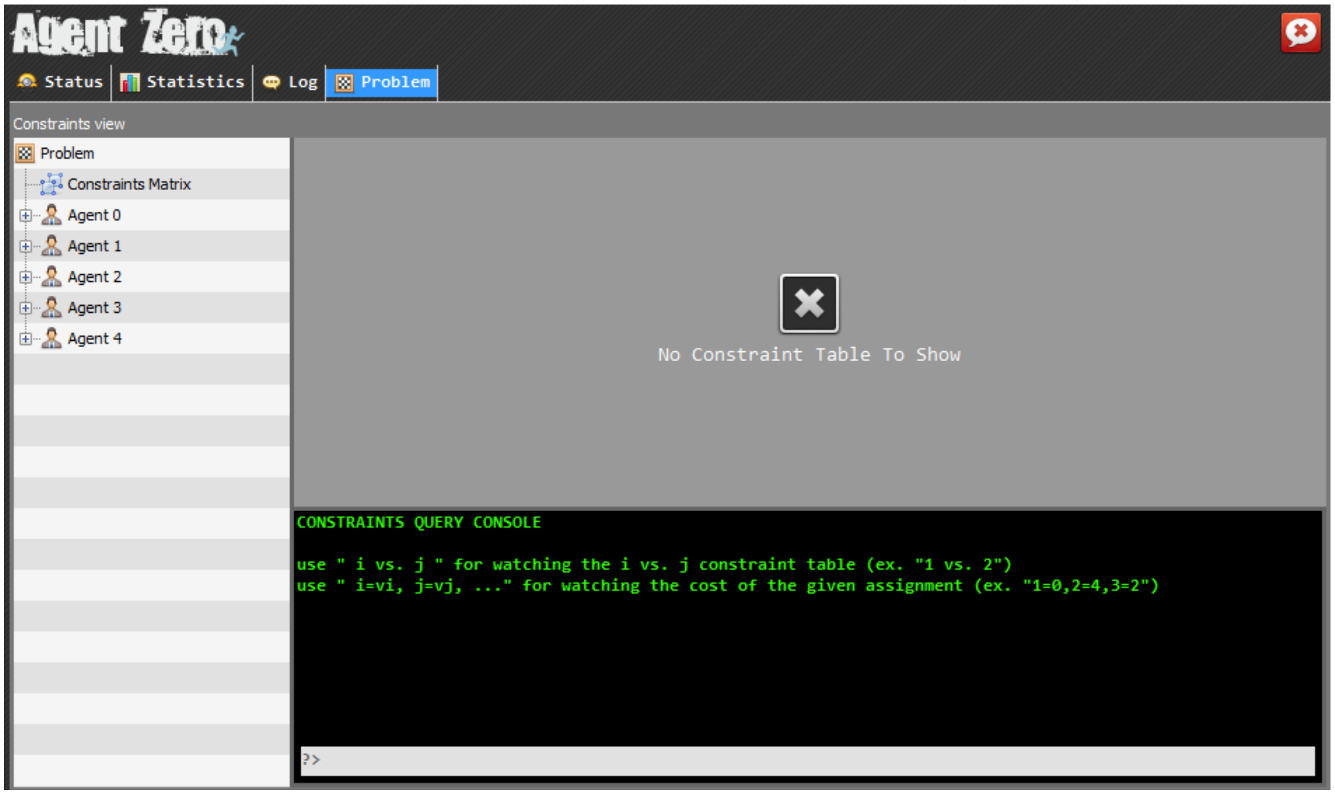
Round Parameters — the set of parameters controlling the execution of the round
+ **name**='round 1' [the round name]
+ **seed**='0' [seed for determining roundiness]
+ **tick-size**='100' [the number of executions in each tick]
+ **run-var**='p2' [the variable to run]
+ **start**='0.1' [starting value of the running variable]
+ **end**='0.9' [ending value of the running variable (explicit)]
+ **tick**='0.1' [the runinig variable value increasment]

Problem Generator — the generator of the problems in this round
+ **n**='5' [number of variables]
+ **d**='5' [domain size]
+ **max-cost**='100' [maximal cost of constraint]
+ **p1**='0.7' [probablity of constraint between two variables]
+ **p2**='0.1' [probablity of conflict between two constrainted variables]

Executed Algorithms — list of the algorithms that participating in this round
+ New

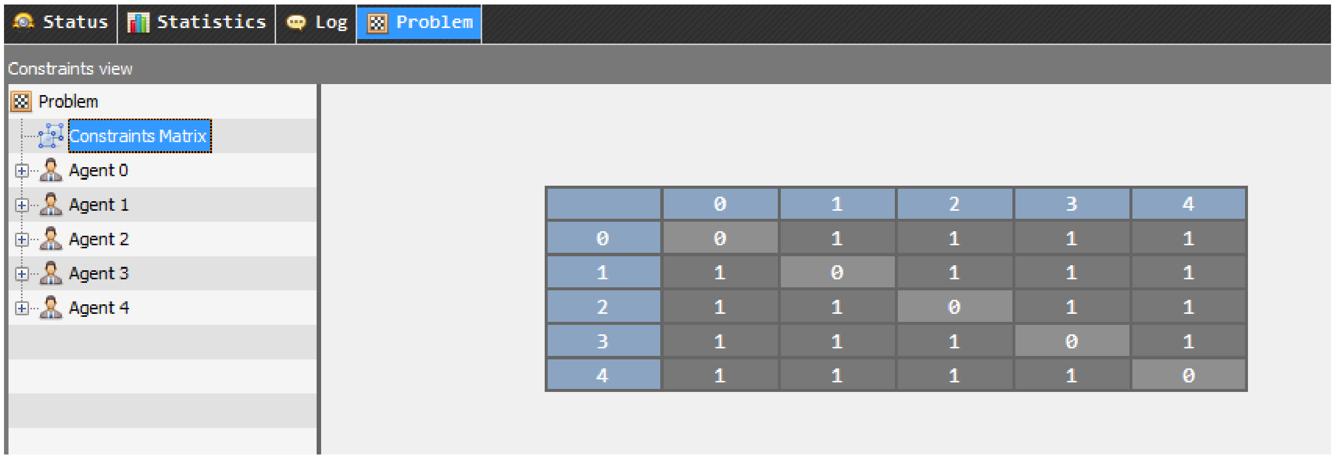
Failure Description — the data about the failure this is whats controlling the debugging
+ **round-name**='round 1' [failing round name]
+ **algorithm-name**='New' [failing algorithm name]
+ **name**='1322678661253' [identifier for this debug info]
+ **seed**='-4962768465676381896' [seed of the failing problem]

After viewing the failure description, if you choose to, press the "Debug This Problem" button in the bottom left corner. You can also delete the selected item by using the swapper button on the top left panel.
Let's press on the "Debug This Problem" button:



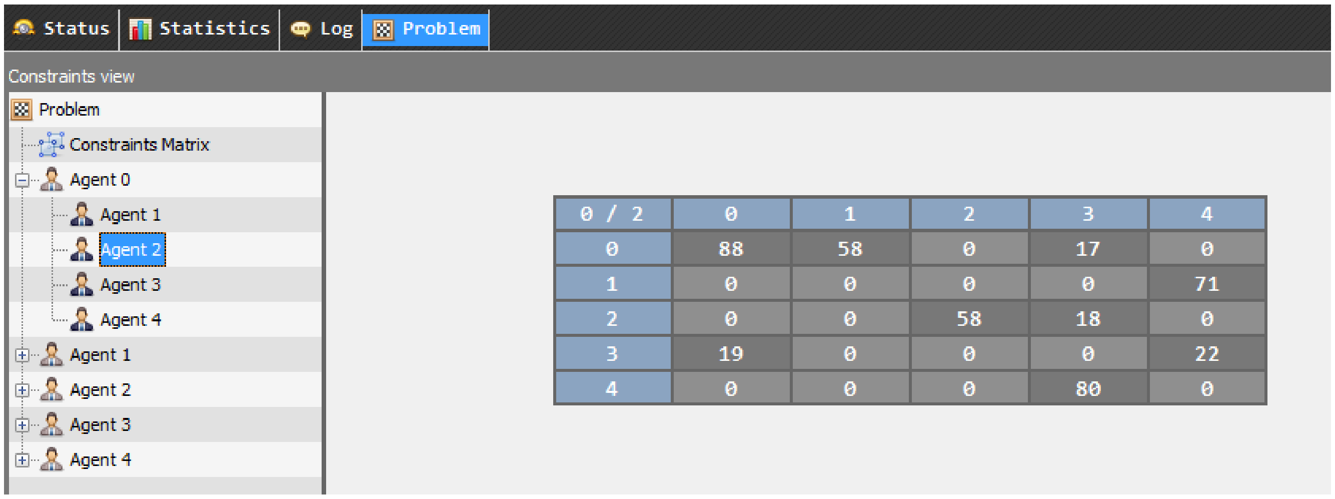
```
CONSTRAINTS QUERY CONSOLE

use " i vs. j " for watching the i vs. j constraint table (ex. "1 vs. 2")
use " i=vi, j=vj, ..." for watching the cost of the given assignment (ex. "1=0,2=4,3=2")

?>
```

A new screen has been added- the problem screen. While debugging, it's occasionally needed to check the debugged problem. On the left – the problem constraint can be chosen and viewed VIA the right panel.
There are two types of nodes in this tree:
The constraints matrix - this matrix shows which agents are constrained:



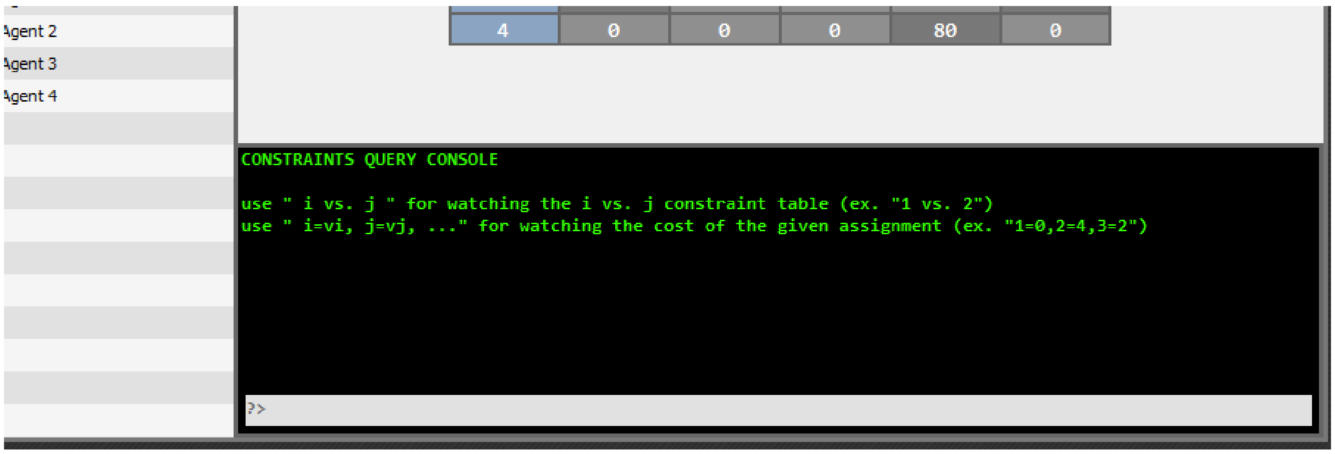
	0	1	2	3	4
0	0	1	1	1	1
1	1	0	1	1	1
2	1	1	0	1	1
3	1	1	1	0	1
4	1	1	1	1	0

The agents - you can view the binary constraint of two agents by selecting them from the tree (if agent pair doesn't exist in the tree – they are not constrained):



0 / 2	0	1	2	3	4
0	88	58	0	17	0
1	0	0	0	0	71
2	0	0	58	18	0
3	19	0	0	0	22
4	0	0	0	80	0

It is also useful to check the cost of an assignment – this screen comes with simple query console for this type of queries:



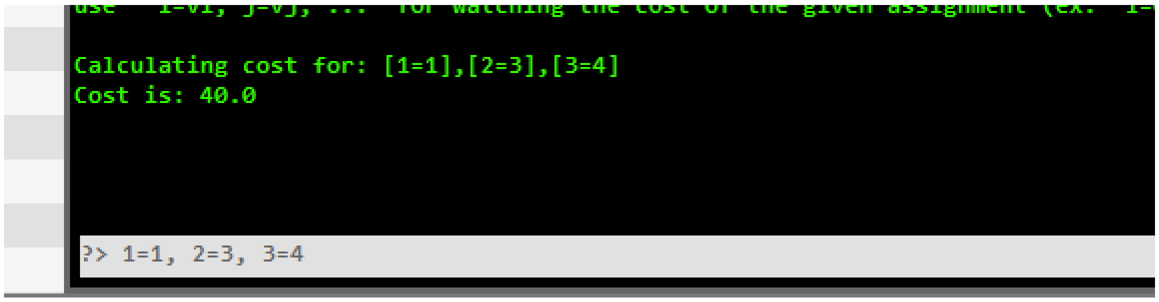
4	0	0	0	80	0

```
CONSTRAINTS QUERY CONSOLE

use " i vs. j " for watching the i vs. j constraint table (ex. "1 vs. 2")
use " i=vi, j=vj, ..." for watching the cost of the given assignment (ex. "1=0,2=4,3=2")

?>
```

As you can see from the console initial output, two types of queries are currently supported:

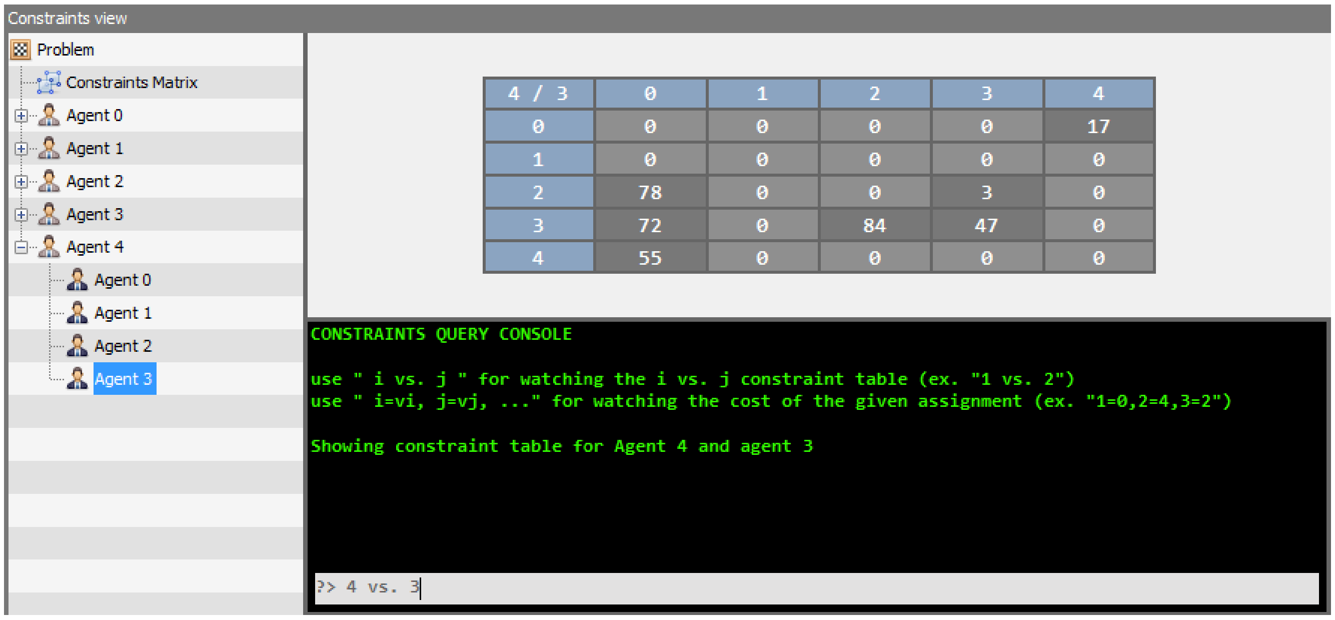
The assignment query - if you want to check the cost of assigning 1=1, 2=3, 3=4

you can just press this expression directly on the console and the calculated cost will be written:



```
Calculating cost for: [1=1],[2=3],[3=4]
Cost is: 40.0

?> 1=1, 2=3, 3=4
```

The vs. query - if there are many agents in the tree and you are too lazy to start searching it for agent 104 vs. agent 22 constraint matrix (we know we are…) you can just press 104 vs. 22 and the required constraint matrix will appear:



4 / 3	0	1	2	3	4
0	0	0	0	0	17
1	0	0	0	0	0
2	78	0	0	3	0
3	72	0	84	47	0
4	55	0	0	0	0

```
CONSTRAINTS QUERY CONSOLE

use " i vs. j " for watching the i vs. j constraint table (ex. "1 vs. 2")
use " i=vi, j=vj, ..." for watching the cost of the given assignment (ex. "1=0,2=4,3=2")

Showing constraint table for Agent 4 and agent 3

?> 4 vs. 3
```

Section 3: working with synchronized search

In this section we are going to learn how to write and execute synchronized search algorithms.

First we will learn about the synchronized execution mode and the basics that the algorithm implementer needs to know in order to successfully write an algorithm that can be executed in this mode, then we will see how to start an agent in synchronized execution mode.

At the end of this section we will inspect a DSA implementation that is written in agent Zero.

Chapter 1: The synchronized execution mode

Agent zero allows an agent to be executed in synchronized execution mode.

At this mode the agents will work synchronically between “shared ticks”.

Each “tick” represents a synchronization point – agents will always receive messages that were sent to them during the previous tick. When agents send messages they will arrive to their destination in the next tick.

A tick ends when all the agents finish receiving the messages that was sent to them during the previous tick (the first tick (=tick 0) is the initializing tick; at this tick the agents “start” function will get called).

The number of ticks that were passed since the algorithm started called “system time”.

Every time a tick ends the method Agent.onMailBoxEmpty will get called.
You should override this method if you want to get notifications about this event (and most of the time this is what you want).

As always, messages are translated to remote procedure calls – you should write the algorithm regularly as you would write any agent zero algorithms (using the @WhenReceived annotation).

If you want to read the current system time you can call the Agent. getSystemTime method – notice that calling this method from within Agent.onMailBoxEmpty will retrieve the last system time (that just means that the function Agent.onMailBoxEmpty is getting called before the time is increased).

That’s about it; you now know everything you need in order to write a synchronized executed agent!

Don’t forgot to run your algorithm in sync-test and not async-test while in development mode

Chapter 2: Implementation example

Let's take a look on a real ADCOP, Synchronized executed algorithm written in Agent Zero:
the highlighted lines demonstrates the concepts discussed in the previous page.

```
@Algorithm(name = "DSA",
 searchType = SearchType.SYNCHRONIZED,
 problemType = ProblemType.ADCOP)
public class DSAAgent extends SimpleAgent {

    private Assignment values;
    private double p;

    @Override
    public void start() {
        values = new Assignment();
        p = 0.5;
        int value = random(this.getDomain());
        this.submitCurrentAssignment(value);
        send("ValueMessage", value).toNeighbores(this.getProblem());
    }

    @WhenReceived("ValueMessage")
    public void handleValueMessage(int value) {
        values.assign(getCurrentMessage().getSender(), value);
    }

    @Override
    public void onMailBoxEmpty() {
        final long systemTime = getSystemTime();

        if (systemTime + 1 == 20000 && isFirstAgent()) {
            finishWithAccumulationOfSubmitedPartialAssignments();
        }

        Integer newValue = calcDelta();
        if (Math.random() > p && newValue != null) {
            submitCurrentAssignment(newValue);
            send("ValueMessage", newValue).toNeighbores(this.getProblem());
        }
    }

    private Integer calcDelta() {
        int ans = this.getSubmitedCurrentAssignment();
        double delta = this.values.calcAddedCost(this.getId(), ans, this.getProblem());
        double tmpDelta = delta;
        for (Integer i : this.getDomain()) {
            double tmp = this.values.calcAddedCost(this.getId(), i, this.getProblem());
            if (tmp < tmpDelta) {
                tmpDelta = tmp;
                ans = i;
            }
        }
        if (delta == tmpDelta) {
            return null;
        }
        return ans;
    }
}
```

SECTION 4: NESTED AGENTS

Sometimes the algorithm that you will implement will need to run other algorithm and work with its results.

The best example of this kind of algorithms is the Pseudo Tree algorithms family.
Such an algorithm will first need to run some kind of Tree Analyzing algorithm (mostly DFS), get its result and use it in its execution.

Nested agents feature designed to solve this need in a simple but a powerful way.

The following section will cover this feature, the usage of a ready-made nested agents and how to build your own nested agent tool.

Chapter 1: Nested agent usage basics

Let's assume that we want to write an algorithm from the Pseudo Tree family – for example DPOP.

Let's see how the code for agent of such an algorithm should look like:

DPOPAgent should first run Distributed DFS to map the problem pseudo tree, the Distributed DFS (DDFS) algorithm should first run Distributed Leader Selection Algorithm (DLSA), then, when the DLSA will complete the DDFS will use its results in order to produce a pseudo tree and when he will complete, the DPOP agent will use the resulted tree in order to produce an assignment.

This means that the DPOP Agent should contain logic of 2 more algorithms, it should also contain a logic that produces synchronization with other agents which are running those algorithms so all of them will move from algorithm to algorithm as a group, or else messages that belong to one algorithm will get sent to other one.

This is also not so modular – if someone else already writes a DLSA you will not be able to use it without fully modifying your agent and copy its code in it.

And we didn't even start talking about debugging such an algorithm – if there is a problem, in which algorithm it happened?

The solution: **Nested Agents**

Nested agents feature allows you to separate your algorithm code into different agents and then combine them anyway you like. For example when the previously discussed DPOP Agent will want to calculate a DFS Tree all he have to do is:

```
Tree tree = new PseudoTree();
tree.calculate(this).andWhenDoneDo(new Continuation() {
        @Override
        public void doContinue() {
           //WHAT TO DO WHEN TREE CALCULATION IS OVER
        }});
```

What just happened?

Tree is a tool that contains a nested agent named DFSAgent.
* this type of tools called nesteable tools.

When calling calculate(this) the agent is actually transforming to DFSAgent and starts its execution.
It is like the DFSAgent is nested inside DPOPAgent (hence the name…)

The DFSAgent messages will arrive only to other DFSAgents, so DPOP is safe from getting unknown messages accidently.

the DFSAgent itself will have the code:

```
LSA lsa = new PseudoTree();
lsa.calculate(this).andWhenDoneDo(new Continuation() {
        @Override
        public void doContinue() {
            //WHAT TO DO WHEN LEADER SELECTION ALGORITHM IS OVER
        }});
```

And now the DFSAgent will get nested and the LSAAgent will became the active Agent.

You will notice that we are not actually instantiating a new agent, instead we are instantiating the result of the wanted algorithm and then calling calculate – this reflects the algorithm need – the DPOP Agent doesn't care how the tree will get calculated, he only care about the final result.

You will also notice that we are supplying the nesteable tool with an implementation of a continuation object; this is just java way to send a callback function (actually a callback closure…).

The last thing that you (as a user of nesteable tool) have to know is what exactly happened when you run calculate –

First the calculation will start the nested agent (calling its start function), then you will return back to the calling function, finish it and from now on (until the nested agent finish) your agent will behave as the nested agent. When the nested agent will finish your continuation function will get called.

This is very important to understand - the calculate method is a non-blocking one!
The calling agent will have to complete its method (start / message handling method) and only then he will became the nested agent – so don't put any code that uses the nesteable tool after the calculation – put it inside the continuation function.

Chapter 2: Writing your own nesteable tool

We saw in the previous chapter how to use readymade nesteable tool, but what if we want to create our own implementation of Leader Selection Algorithm?

In that case we will want to build new nesteable tool – this is fairly simple:

Just create a new class (let’s assume you called it MyLSA) with all the needed fields for the result of this algorithm (in that case we will have a field: selectedLeader).
Next, you should write an inner class that extend SimpleAgent (let’s assume you called it MyLSAAgent).
This class will have access to the field selectedLeader as it is an inner class of MyLSA.
Write in this class the LSA logic like any other agent, and finally implement the NesteableTool.createNestedAgent function to return new MyLSAAgent().

That’s it, here is a simple template following those steps:

```java
public class MyLSA extends NesteableTool {

    private int selectedLeader;

    @Override
    public Integer getSelectedLeader() {
        return this.selectedLeader;
    }

    @Override
    protected SimpleAgent createNestedAgent() {
        return new MyLSAAgent();
    }

    public class MyLSAAgent extends SimpleAgent {

        @Override
        public void start() {
            …
        }

        .
        .
        .
        @WhenReceived("SomeMessage")
        public void handleSomeMessage(int leader) {
            selectedLeader = leader;
            finish();
        }

    }
}
```

Section 5: Problem Generators

Agent Zero provides several problem generators for your algorithm to work with.
Occasionally you will want to write your own problem generator that can generate structured problems and then test your algorithm performance upon these problems.

In this section we will learn how to write our own problem generator.
To do so we will examine an existing problem generator code – the dcsp-unstructured generator:

```
@Register(name="dcsp-unstructured")
public class UnstructuredDCSPGen extends AbstractProblemGenerator {

    @Variable(name = "n", description = "number of variables")
    int n = 2;
    @Variable(name = "d", description = "domain size")
    int d = 2;
    @Variable(name = "p1", description = "probability of constraint between two variables")
    float p1 = 0.6f;
    @Variable(name = "p2", description = "probability of conflict between two constrained variables")
    float p2 = 0.4f;


    @Override
    public void generate(Problem p, Random rand) {
        p.initialize(ProblemType.DCSP, n, new ImmutableSet<Integer>(Agt0DSL.range(0, d - 1)));
        for (int i = 0; i < p.getNumberOfVariables(); i++) {
            for (int j = 0; j < p.getNumberOfVariables(); j++) {
                if (rand.nextDouble() < p1) {
                    buildConstraint(i, j, p, rand);
                }
            }
        }
    }

    private void buildConstraint(int i, int j, Problem p, Random rand) {
        for (int vi = 0; vi < p.getDomain().size(); vi++) {
            for (int vj = 0; vj < p.getDomain().size(); vj++) {
                if (i == j) {
                    continue;
                }
                if (rand.nextDouble() < p2) {
                    final int cost = rand.nextInt(2);
                    p.setConstraintCost(i, vi, j, vj, cost);
                    if (sym) {
                        p.setConstraintCost(j, vj, i, vi, cost);
                    }
                }
            }
        }
    }
}
```

Let’s examine parts of this class:

Part 1: **Registration**:

```
@Register(name="dcsp-unstructured")
public class UnstructuredDCSPGen extends AbstractProblemGenerator { …
```

AgentZero will need to know that you wrote external module - all you need to do in order to load your module is to give it a name using the `@Register` annotation.
In order for AgentZero to know that you are building a problem generator your class must extend `AbstractProblemGenerator` class.

Part 2: **Variables:**

```
…
    @Variable(name = "n", description = "number of variables")
    int n = 2;
    @Variable(name = "d", description = "domain size")
    int d = 2;
    @Variable(name = "p1", description = "probability of constraint between two variables")
    float p1 = 0.6f;
    @Variable(name = "p2", description = "probability of conflict between two constrained variables")
    float p2 = 0.4f;
…
```

Most problem generators will need variables in order to build the problem.
A variable type is one of java primitives or String; it is a field in the problem generator class with public/protected or default access modifier and the `@Variable` annotation with name and description attributes.

Part 3: **implementing the Generate method**

```
…
    @Override
    public void generate(Problem p, Random rand) {
        p.initialize(ProblemType.DCSP, n, new ImmutableSet<Integer>(Agt0DSL.range(0, d - 1)));
        for (int i = 0; i < p.getNumberOfVariables(); i++) {
            for (int j = 0; j < p.getNumberOfVariables(); j++) {
                if (rand.nextDouble() < p1) {
                    buildConstraint(i, j, p, rand);
                }
            }
        }
    }
…
```

Implementing the AbstractProblemGenerator interface requires you to implement the generate function.

This function receives uninitialized problem instance as argument and a seeded random object for your use.
The concept is pretty simple: before this function is called all the generator defined variables are filled by AgentZero. Then, AgentZero will generate a random object and a problem and call your generate function.
The first thing your generate function should do is initialize the problem supplying its type, number of variables and domain values.
Next, you should add constraints to the problem by calling the problem’s setConstraint function (If you need to use random values please use the provided random object to generate them).
The usage of this object will give AgentZero the opportunity if needed, to regenerate the same problem by providing you with the same random object.

SECTION 6: STATISTIC ANALYZERS

Statistic analyzer is a tool that allows us to collect and analyze the statistical information regarding the experiment. Currently, AgentZero provides 3 basic statistic analyzers:
- nccc-sc : number of concurrent constraint checks
- ncsc-sc : number of concurrent steps of computation
- cc-sc : number of constraint checks

If you want to collect some statistic that is not implemented yet, you need to create your own statistic analyzer.

Now we will show an example of a statistic analyzer:

```
@Register(name="cc-sc")
public class CCStatisticCollector extends AbstractStatisticCollector<CCStatisticCollector.CCRecord> {

    @Override
    public VisualModel analyze(Database db, Test r) {
        String query = "select AVG(cc) as avg, rVar from CC where ROUND = '"+r.getName()
                                                     +"' group by rVar order by rVar";
        LineVisualModel line = new LineVisualModel(r.getRunningVarName(), "Avg(CC)", "CC");
        ResultSet rs = db.query(query);
        while (rs.next()) {
            line.setPoint(rs.getFloat("rVar"), rs.getFloat("avg"));
        }
        return line;
    }

    @Override
    public void hookIn(final Agent[] agents, final Execution ex) {
        agents[0].hookIn(new BeforeCallingFinishHook() {

            @Override
            public void hook(Agent a) {
                int sum = 0;
                for (Agent ag : agents) {
                    sum += ag.getNumberOfConstraintChecks();
                }
                submit(new CCRecord(ex.getRound().getCurrentVarValue(), sum));
            }
        });
    }

    @Override
    public String getName() {
        return "Constraint Checks";
    }


    public static class CCRecord extends DBRecord {

        float rVar;
        int cc;

        public CCRecord(float rVal, int cc) {
            this.rVar = rVal;
            this.cc = cc;
        }

        @Override
        public String provideTableName() {
            return "CC";
        }
    }
}
```

Let's go over the implementation:

Part 1: **Registration**:

```
@Register(name="cc-sc")
```

First, name your statistic analyzer and register it in the system using the Register annotation (in the example @Register(name="cc-sc")).

Part 2: **Creating the inner class for the record:**

…
```
    public static class CCRecord extends DBRecord {

        float rVar;
        int cc;

        public CCRecord(float rVal, int cc) {
            this.rVar = rVal;
            this.cc = cc;
        }

        @Override
        public String provideTableName() {
            return "CC";
        }
    }
```
…

All the statistic information will be stored in a data base and you have to decide how your record will be saved. In order to do this, implement an inner class that extends DBRecord. All you need to do is to determine what fields (types and names) it must have (every field equals to a column in the table).Now implement a default constructor and the method provideTableName, which returns the name you want to give to the table.

Part 3: **Extending AbstractStatisticCollector:**

```
public class CCStatisticCollector extends AbstractStatisticCollector<CCStatisticCollector.CCRecord> {
```
…

Now, we go back to the statistic analyzer class. The class you create needs to extend AbstractStatisticCollector<T>, where T is your created inner class which extends DBRecord.
you need to implement 3 methods: getName(returns the name of the statistic analyzer), HookIn and Analyze.

Part 4: **Implementing the HookIn method:**

…
```
@Override
    public void hookIn(final Agent[] agents, final Execution ex) {
        agents[0].hookIn(new BeforeCallingFinishHook() {

            @Override
            public void hook(Agent a) {
                int sum = 0;
                for (Agent ag : agents) {
                    sum += ag.getNumberOfConstraintChecks();
                }
                submit(new CCRecord(ex.getRound().getCurrentVarValue(), sum));
            }
        });
    }
```
…

This method is the statistics collector. The method registers listener for method calls made by the agent. When the agent calls a method that a hook is registered to, the agent statistic will be retrieved and submitted to the data base. It is important to note that the HookIn will be called before every execution in order to allow the statistic to be initialized, this happens because the statistic analyzer is not created for each execution.

Part 5: **Implementing the Analyze method:**

```
…
@Override
    public VisualModel analyze(Database db, Test r) {
        String query = "select AVG(cc) as avg, rVar from CC where ROUND = '"+r.getName()
                                                         +"' group by rVar order by rVar";
        LineVisualModel line = new LineVisualModel(r.getRunningVarName(), "Avg(CC)", "CC");
        ResultSet rs = db.query(query);
        while (rs.next()) {
            line.setPoint(rs.getFloat("rVar"), rs.getFloat("avg"));
        }
        return line;
    }
…
```

This method is the actual analyzing process. You need to write a SQL query which will summarize the data you collected in the table. Then, you create the graph model you want to show the data in. You get the ResultSet which returns from the query, fill in the model with the data and return the model (In the example, we create a line graph model).

That’s it. AzentZero will do the rest for you.

SECTION 7: VARIABLES IN AGENTS

If you ever encounter a situation when you want to control the algorithm execution based on an external variable – for example you wrote your own cool optimization to algorithm ALG and want to test ALG performance without the optimization vs. with the optimization.

Instead of copying ALG code to new ALG_BETR agent you can add a Boolean external variable B to your ALG agent so that if the B is true the optimization will get turned on. Then you will be able to test ALG with B=false vs ALG with B=true.
This can be done pretty simply by using the @Variable annotation – for example:

```
public class ALGAgent extends SimpleAgent{
    @Variable(name = "B", description="turning on my awesome optimization")
    boolean b = false;
...
}
```

Agent zero will assign all your variables before calling to the start function.
If you want in development time to assign a value to your variable you can do this in the following way:

```
<algorithm name=”ALG”>
   <assign var=”B” val=”true”/>
</algorithm>
```

That’s it – pretty simple…

Section 8: Message Delays

Message Delay is a feature meant to mimic the network effect on the algorithms performance by different aspects .

Agent Zero came bundled with a default message delayer that mimics the behavior of TCP and adds delay that is measured by one of the types: NCCC, NCSC , the delays are added randomly and the user can select the minimum and maximum delay

Message delay can only be applied on asynchronous execution.

To activate message delay feature you should add message delayer to your execution, you can do this easily by adding it in test.xml:

```
…
<async-test …>
        …
        <default-message-delayer  type=”NCCC” seed=”42” maximum-delay=”100” minimum-delay=”0” />
…
```

A user can also create new message delayer by implementing the message delayer interface.

Let’s look at the default implementation of message delayer to understand the concepts of creating a new one:

```
@Register(name = "default-message-delayer")
public class DefaultMessageDelayer implements MessageDelayer {

    @Variable(name = "type", description = "the type of the delay: NCCC/NCSC", defaultValue = "NCCC")
    String type = "NCCC";
    @Variable(name = "seed", description = "seed for delay randomization", defaultValue = "42")
    int seed = 42;
    @Variable(name = "maximum-delay", description = "the maximum delay", defaultValue = "100")
    int maximumDelay = 100;
    @Variable(name = "minimum-delay", description = "the minimum delay", defaultValue = "0")
    int minimumDelay = 0;

    int[][] previousTime;
    Random rnd = null;

    @Override
    public int getInitialTime() {
        return 0;
    }

    @Override
    public int extractTime(Message m) {
        if (type.equals("NCCC")) {
            return ((Long) m.getMetadata().get("nccc")).intValue();
        } else {
            return ((Long) m.getMetadata().get("ncsc")).intValue();
        }
    }

    @Override
    public void addDelay(Message m, int from, int to) {
        int delay = rnd.nextInt(maximumDelay - minimumDelay) + minimumDelay;
        int ntime = Math.max(delay + previousTime[from][to], extractTime(m) + delay);
        previousTime[from][to] = ntime;
        if (type.equals("NCCC")) {
            m.getMetadata().put("nccc", (long)ntime);
        } else {
            m.getMetadata().put("ncsc", (long)ntime);
        }
    }

    @Override
    public void initialize(Execution ex) {
        int n = ex.getGlobalProblem().getNumberOfVariables();
        previousTime = new int[n][n];
        rnd = new Random(seed);
    }
}
```

as you can see, there are only 4 main functions to implement:

- `getInitialTime: this function should return the initial time before the execution even started`
- `extractTime: this function received a message and extract the time from this message`
- `addDelay: this function add delay on the time field in the message`
- `initialize: this function is called before each execution (algorithm that is executed)`
  `this is very similar to a constructor call - as always, modules should not define a constructor - the module should reinitialize itself relative to the given execution every time this function called.`