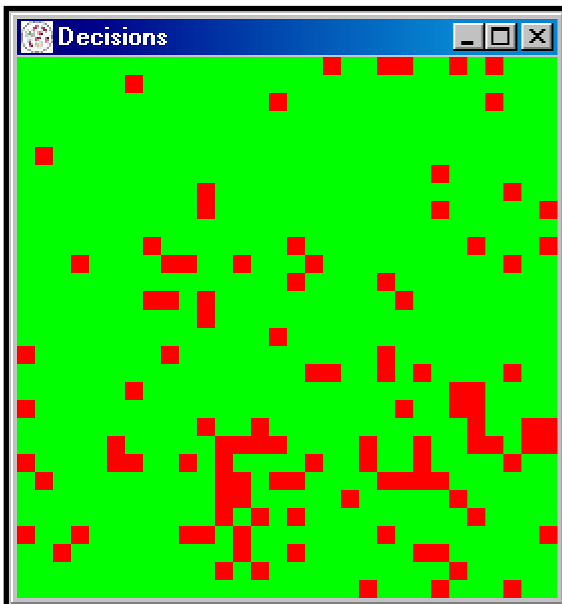
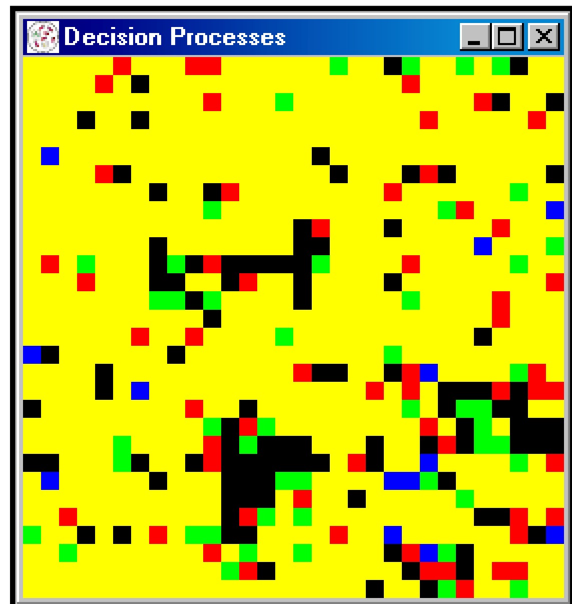
CASD
Model 0

Nick Gotts, Luis Izquierdo, Gary Polhill
Macaulay Institute
Craigiebuckler, Aberdeen AB15 8QH
United Kingdom





User Guide

TABLE OF CONTENTS

1. INTRODUCTION 1

2. THE CONCEPTUAL MODEL 1

2.1. THE GAME 1

2.2. THE PLAYERS: DESIGN OF A GENERAL CASE-BASED REASONER 2

2.2.1. Simpler Agents: particular cases of the general design 4

3. MODEL IMPLEMENTATION: THE PROGRAM 6

3.1. INSTALLATION 6

3.2. PARAMETERS OF THE PROGRAM 6

3.2.1. Spatial parameters 6

3.2.2. Case-base parameters 8

3.2.3. Dimensions of utility's Thresholds 8

3.2.4. Structure of the game played by the agents 8

3.3. HOW TO USE THE PROGRAM 9

3.3.1. Parameter Files 9

3.3.2. Usage synopsis and command-line options 10

3.3.3. Graphical User Interface (GUI) Mode 12

3.3.4. Batch mode 15

3.3.5. Analysing thousands of runs: Scripts 15

3.4. HOW TO EXTEND THE PROGRAM: IMPLEMENT YOUR OWN GAMES 17

1. Introduction

This document outlines the usage of CASD (Casuistry And Social Dilemmas) model 0. CASD is an agent-based model designed to study the outcome of games played by agents that use a form of case-based reasoning in their decision-making algorithm. The code is written in Objective-C, and is known to work with Swarm version 2.1.1 on a PC running Windows 2000 and XP, and with Swarm testing 2001-12-18 on a Sun running Solaris 8.

2. The conceptual model

In CASD there are a number of players (agents) who play a game repeatedly according to the following schedule of events:

1. Agents decide whether to cooperate or defect.
2. The payoff is calculated for each agent; agents' wealth is updated accordingly.
3. Agents are given the opportunity to disapprove of their neighbours.
4. Agents with negative wealth leave the simulation. New agents come into play to substitute for those who leave.

2.1. The game

There are two games that are already implemented in CASD: "Symmetric2x2" and the "Tragedy of the Commons game":

Symmetric2x2: this game can be played only by 2 players. Table 1 shows the Payoff matrix for the Symmetric2x2 game structure. Payoffs on the bottom left of each cell are for Player 1 and payoffs on the top right are for Player 2.

		Player 2	
		Cooperate	**Defect**
Player 1	**Cooperate**	***Reward*** ***Reward***	***Temptation*** ***Sucker***
	Defect	***Sucker*** ***Temptation***	***Punishment*** ***Punishment***

Table 1 — Payoff matrix for Symmetric2x2.

This structure allows us to model, for example, the following games (payoffs are denoted by its initial letter):

Prisoner's Dilemma:	T > R > P > S;
Chicken:	T > R > S > P;
Stag Hunt:	R > T > P > S;
Battle of the Sexes:	S > T > R > P;
Leader:	T > S > R > P;

The "Tragedy of the Commons game" (TC): this game can be played by any number of players. In this game every player gets a reward as long as there are no more than *M* defectors. Table 2 shows the Payoff matrix for a particular player.

	Fewer than *M* others defect	*M* others defect	More than *M* others defect
Player cooperates	***Coop-P* + *Reward-P***	***Coop-P* + *Reward-P***	***Coop-P***
Player defects	***Def-P* + *Reward-P***	***Def-P***	***Def-P***

Table 2 — Payoff matrix of the "Tragedy of the Commons game" for a particular player.

2.2. The players: design of a general case-based reasoner

This section describes the design of Agents which use a simple form of Case-Based Reasoning (CBR) to decide whether to cooperate or defect. CBR basically consists of solving a problem by remembering a previous similar situation and by reusing information and knowledge of that situation.

In CBR, a case is a contextualised piece of knowledge representing an experience. In general the experience could be the Agent's own, or a neighbour's. However, in CASD model-0, Agents only use cases that they have experienced themselves.

A case for an Agent, *i.e*. the experience they lived in time-step *t*, comprises:

1. The time-step *t* when the case occurred.

2. The perceived state of the world at the beginning of time-step *t*, characterised by the factors that the Agent considers relevant to estimate the Payoff. These are:

   Descriptor 1: the number of other defectors.
   Descriptor 2: the decision that the Agent made.

   Agents are able to remember *b* time-steps back (*e.g*. if $b = 2$, the state of the world for the Agent will be determined by the number of other defectors and the decisions made, both in time-step t-1 and in time-step t-2).

3. The decision they made in that situation, *i.e*. whether they cooperated or defected in time-step *t*, having observed the state of the world in that same time-step.

4. The outcome after having decided in time-step *t*. The outcome is composed of both the payoff that the Agent obtained and the number of neighbours that disapproved of the Agent in the following *f* time-steps (*e.g*. if $f = 2$, agents will remember the payoff obtained and the number of disapproving neighbours, both in time-step *t* and in time-step t+1).

Thus the case representing the experience lived by Agent *A* in time-step *t* has the following structure:

<table>
<tr><td rowspan="2">t</td><td>df_{t-b} … df_{t-2} df_{t-1}</td><td rowspan="2">d_t</td><td>p_t p_{t+1} … p_{t+f-1}</td></tr>
<tr><td>d_{t-b} … d_{t-2} d_{t-1}</td><td>dn_t dn_{t+1} … dn_{t+f-1}</td></tr>
</table>

where

df_i is the number of defectors (excluding agent *A*) in time-step *i*,
d_i is the decision made by Agent *A* in time-step *i*,
p_i is the payoff obtained by Agent *A* in time-step *i*, and
dn_i is the number of disapproving neighbours in time-step *i*.

The number of cases that Agents can keep in memory is unlimited. It is also worth noting that a case that occurred in time-step t is not available for use until time-step $(t + f)$. Likewise, no cases are available for any Agent until $(b + f)$ time-steps have gone by in the simulation.

Agents make their decision whether to cooperate or not by retrieving two cases: the most recent case which occurred in a *similar* situation for each of the two decisions (each of the two possible values of d_t). A case is perceived by the Agent to have occurred in a *similar* situation if and only if its state of the world is a perfect match with the current state of the world observed by the Agent holding the case. The only function of the state of the world is to determine whether two situations look *similar* to the Agent or not.

In a particular situation (*i.e.* for a given state of the world) an Agent must face one of the following three possibilities:

1. The Agent cannot recall any previous situations that match the current perceived state of the world. In CBR terms, the Agent does not hold any appropriate cases for the current state of the world. In this case the Agent makes an unbiased random decision.

2. The Agent does not remember a previous similar situation when they made a certain decision, but they do recall at least one similar situation when they made the other decision. In CBR terms, all the appropriate cases the Agent recalls have the same value for d_t. In this situation, Agents will explore the non-applied decision if the Payoff they obtained in the last previous similar situation was below their *AspirationThreshold*; otherwise they will keep the same decision they previously applied in similar situations.

3. The Agent remembers at least one previous similar situation when they made each of the two possible decisions. In this situation, the Agent will focus on the most recent case for each of the two decisions and choose the decision that provided them with the higher average payoff calculated using the f payoffs in each case (see figure below). In case of tie, the Agent will make an unbiased random decision.

<table>
<tr><td rowspan="2">t</td><td>df_{t-b} … df_{t-2} df_{t-1}</td><td rowspan="2">d_t</td><td>p_t p_{t+1} … p_{t+f-1}</td></tr>
<tr><td>d_{t-b} … d_{t-2} d_{t-1}</td><td>dn_t dn_{t+1} … dn_{t+f-1}</td></tr>
</table>

There is a meta-rule to decide whether to cooperate or defect: every time the Agent aims to defect in a situation similar to one where the Agent has defected before (*i.e.* points 2 and 3 above), they will check whether the average number of neighbours who disapproved of the Agent the last time they defected in a similar situation (calculated using the *f* numbers of disapproving neighbours in the most recent case that matches the state of the world - see figure below) is higher than the *PeerPressureThreshold*. If it is indeed higher, then the Agent will cooperate. The *PeerPressureThreshold* is the maximum number of disapproving neighbours that an Agent can bear.

<table>
<tr><td rowspan="2">t</td><td>df_{t-b} … df_{t-2} df_{t-1}</td><td rowspan="2">d_t</td><td>p_t p_{t+1} … p_{t+f-1}</td></tr>
<tr><td>d_{t-b} … d_{t-2} d_{t-1}</td><td>dn_t dn_{t+1} … dn_{t+f-1}</td></tr>
</table>

In CASD model 0 cooperators disapprove of their neighbours who are defecting.

All the Agents share the same *AspirationThreshold* (*AT*), the same *Backwards Memory b*, the same *ForwardMemory f*, and the same *PeerPressureThreshold* (*PPT*).

Figure 1 sketches the outlined decision process.

2.2.1. Simpler Agents: particular cases of the general design

***ForwardMemory* = 1**

If *ForwardMemory* equals 1, the structure of the case is the following:

<table>
<tr><td rowspan="2">t</td><td>df_{t-b} … df_{t-2} df_{t-1}</td><td rowspan="2">d_t</td><td>p_t</td></tr>
<tr><td>d_{t-b} … d_{t-2} d_{t-1}</td><td>dn_t</td></tr>
</table>

Agents will not calculate any average, but they will only look at the immediate consequences of their decisions.

***peerPressureThreshold* ≥ number of players – 1**

Setting the *peerPressureThreshold* to any value greater than or equal to the number of players minus one means that Agents will never change their decision because of peer pressure. The social approval meta-rule will not apply in this case.

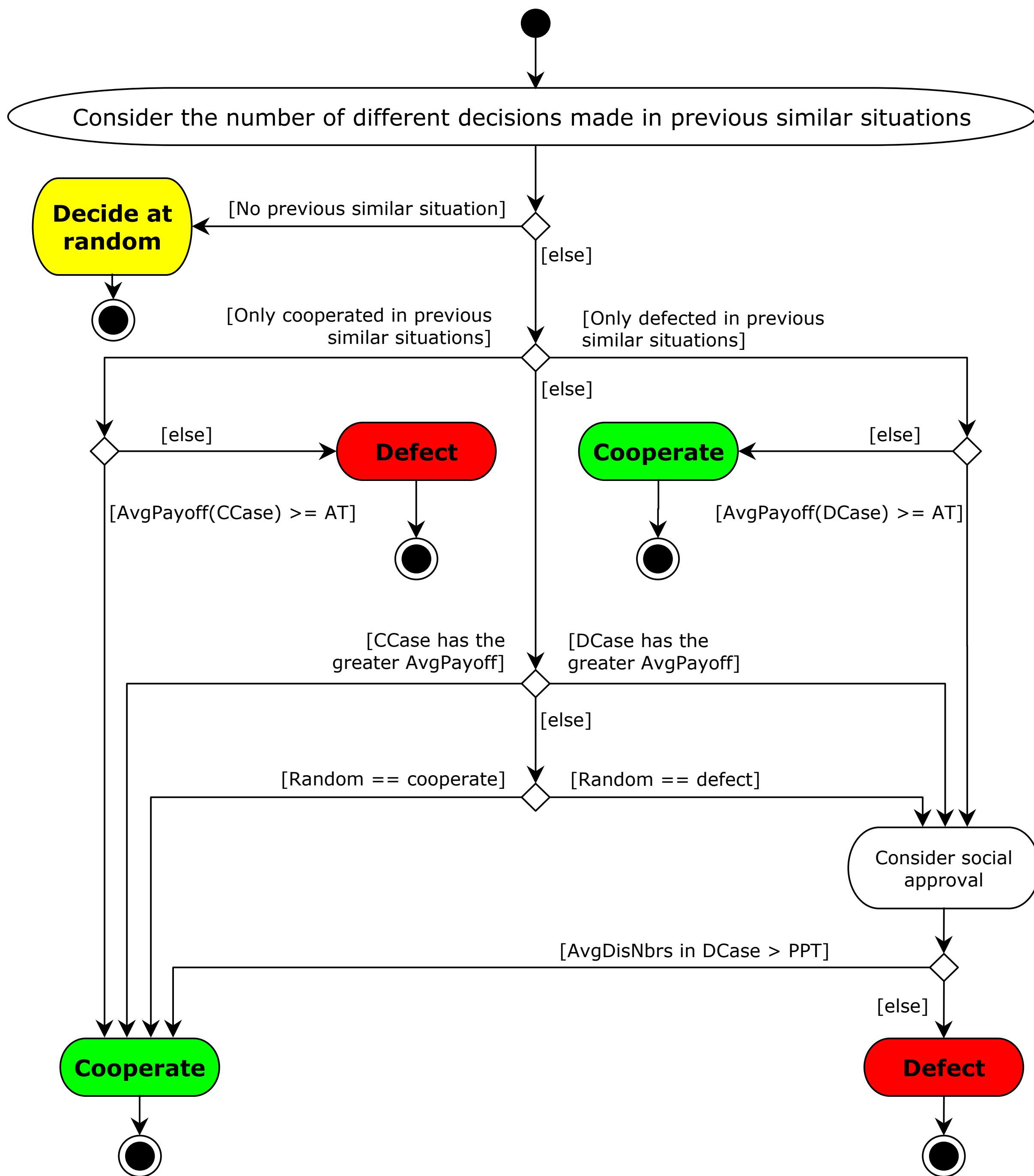


Figure 1 — UML activity diagram of the decision making process used by case-based reasoners in CASD.

***Backwards Memory b* = 0**

If we set the Backwards Memory to zero, every situation will look similar to the Agents, so the analysis of the simulation becomes much easier.

***Forward Memory f* = 0**

If Forward Memory is set to zero, Agents will always decide at random.

3. Model implementation: the program

3.1. Installation

Assuming you have the appropriate version of Swarm installed on your platform, and your environment set up to use it, installation of CASD model 0 involves unpacking the gzipped tar file, and compiling the source code therein from a terminal window in Unix, or the Swarm >> Terminal application in Windows:

```
gunzip –c casd-0.tar.gz | tar xf –
cd CASD-0
make
```

The application should compile successfully without error, creating the executable casd on a Unix platform, or casd.exe on Windows. I have experienced compilation problems when I use WinZip to untar the file `casd-0.tar`, so please use `tar` as indicated above.

CASD takes special care of floating point numbers, following work led by Gary Polhill that showed the nasty effects that floating point numbers can have in Agent-Based simulation. If you use only integers as values for all the parameters you should never suffer floating point errors[1] since the only mathematical operations used in the model are addition, subtraction and multiplication. In any case, if you are on a Unix platform you can check for floating point errors for any value of the parameters. In order to use the full capacity to detect floating point errors implemented in CASD (provided by Gary Polhill), you have to substitute MakefileDoubles for Makefile.

3.2. Parameters of the program

The parameters of the program can be classified into 4 categories:

3.2.1. Spatial parameters

In the implementation of the model, Agents are distributed on a regular bounded rectangular grid of size *xSize* × *ySize* (one Agent per cell). The topology of the grid may be planar or toroidal. In a toroidal topology, edge cells at the North of the grid 'wrap-around' to those at the South (and vice versa), as do those at the East and West. Figure 2 shows the topological effect of wrapping around. In a planar topology, no wrap-around of edge cells takes place.

[1] Except in the very unlikely case that any number in the simulation gets out of the range [-2^{53}, 2^{53}].

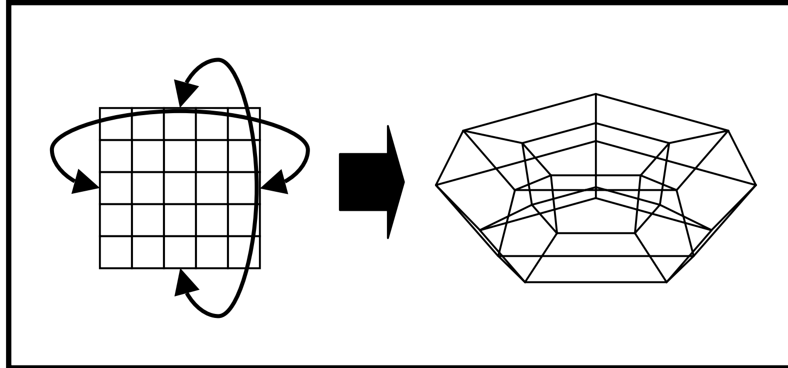


Figure 2 — Wrapping around a grid of cells in two dimensions to form a torus.

The grid is used to define the Agents' set of neighbours using the Moore or the von Neumann neighbourhood function with user-specified *radius* (Figure 3).

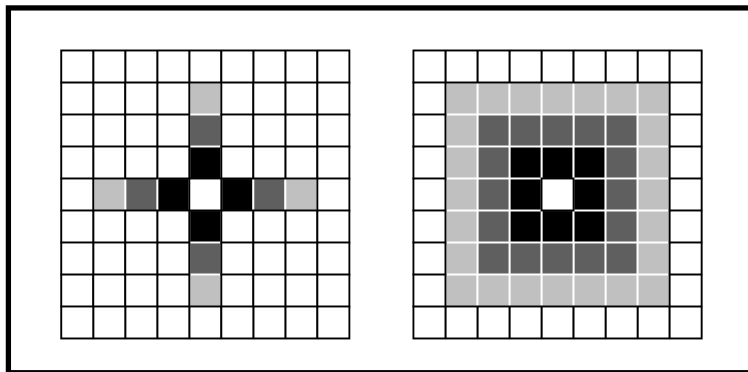


Figure 3 — The effect of neighbourhood radius on von Neumann (left) and Moore neighbourhoods (right). The outermost cells of the neighbourhood of the centremost cell is shown for a neighbourhood radius of 1 (black), 2 (dark grey) and 3 (light grey).

The spatial parameters in the program are the following:

`XSize`	{natural number}
`YSize`	{natural number}
`envShape`	{planar, toroidal}
`nbrhood`	{moore, von-neumann}
`radius`	{natural number}

3.2.2. Case-base parameters

Parameter	Description
bMemory	Backwards Memory {natural number + 0}
fMemory	Forward Memory {natural number + 0}
descriptorOtherDefectorsStr	Use the number of neighbouring defectors as a descriptor of the agents' state of the world {YES, NO}
descriptorMyDecisionsStr	Use the agent's own decisions as a descriptor of their state of the world {YES, NO}

3.2.3. Dimensions of utility's Thresholds

Parameter	Description
expThreshold	Aspiration Threshold. When recalling only similar situations in which the Agent made just one certain decision, if the expected payoff of that decision equals or exceeds the Aspiration Threshold, the decision is kept without exploring the other one. {Double}
peerPressureThreshold	Maximum number of neighbours that an Agent is prepared to be disapproved of. In other words, the maximum number of disapproving neighbours that an Agent can bear. If the expected number of disapproving neighbours is higher than the Peer Pressure Threshold, the Agent will always cooperate (the peer pressure is too high, if you like), no matter the expected payoff. {integer}

3.2.4. Structure of the game played by the agents

Parameter	Description
payoffCalClassStr	This is a class (which must conform to PayoffCalculator protocol) that sets the payoffs for every possible situation. So far, the Symmetric2x2 game and the Tragedy of the Commons game are implemented.
payoffParameterFile	This is a parameter file for the payoffCalClassStr class.

3.3. How to use the program

3.3.1. Parameter Files

1. Edit a main parameter file, such as the following:

```
(list
 (cons 'modelSwarm
       (make-instance 'ModelSwarm
                      #:bMemory 1
                      #:fMemory 1
                      #:descriptorOtherDefectorsStr YES
                      #:descriptorMyDecisionsStr YES
                      #:expThresholdStr DoubleSimple=20.0
                      #:peerPressureThreshold 1
                      #:envShape planar
                      #:nbrhood moore
                      #:xSize 1
                      #:ySize 2
                      #:radius 1
                      #:payoffCalClassStr Symmetric2x2
                      #:payoffParameterFile PD.scm)))
```

2. Edit a payoff parameter file (whose name is the argument of payoffParameterFile in the main parameter file) for the payoffCalClassStr payoff structure. Examples are provided below:

For the Prisoners' Dilemma:

```
(list
 (cons 'payoffCalculator
       (make-instance 'Symmetric2x2
                      #:temptationStr DoubleSimple=10.0
                      #:rewardStr DoubleSimple=9.0
                      #:punishmentStr DoubleSimple=2.0
                      #:suckersStr DoubleSimple=1.0)))
```

For the Tragedy of the Commons game:

```
(list
 (cons 'payoffCalculator
       (make-instance 'Reward
                      #:defectionYieldStr DoubleSimple=10.0
                      #:cooperationYieldStr DoubleSimple=9.0
                      #:rewardStr DoubleSimple=10.0
                      #:rewardIfThisOrFewer 15)))
```

3.3.2. Usage synopsis and command-line options

In the usage synopsis below, square brackets [ ] are used to denote an optional element, and angle brackets < > to describe some value the user should provide.

```
./casd [+v <verbosityLevel>] [+V <verbosityFile>] [+t <stopTime>]
[+c <checkTime>] [+d <decisionsString>] [+b] <parameterFile> [-s]
[-S <seed>]  [-b]  [other Swarm flags]
```

Please note that the + options must appear first after the command and before the parameter file, whereas the – options must appear after the parameter file. As we can see above, the only indispensable requirement is to specify the main parameter file.

The command-line options are explained below.

+v <verbosityLevel> takes an integer argument specifying the level of verbosity (higher numbers mean more messages, according to the verbosity file)

+V <verbosityFile> takes a filename argument. This file is called the 'verbosity file'. It will have to contain a list of message types that are generated by the system, and a verbosity level at or above which the messages are to be printed. The default 'verbosity file' is casd.verby. An example of how this file looks like is the following:

```
ParseArgs 2
Progress 2
StateUpdate 5
AgentsUpdatingHistory 5
CaseFound 10
Cycles 2
AgentsResetting 2
PayoffCalcUpdatingDecisions 10
PayoffCalcGivingPayoffs 10
AgentsDeciding 10
AgentsRemembering 10
AgentsStateUpdate 10
AgentsDecidingAtRandom 4
Equilibria 1
NumberRandomDecisions 10
AgentsJudging 10
CyclicBehaviour 1
AgentsDecidingDeterministically 10
NumberDeterministicDecisions 10
AgentsUpdatingAge 10
```

For example, using the verbosity file above, if the verbosity level set after the option +v is 2, the following type of messages will be printed: *ParseArgs*, *Progress*, *Cycles*, *AgentsResetting*, *Equilibria*, and *CyclicBehaviour*.

+t <stopTime> takes an integer argument specifying the stop time-step. The simulation will stop at time-step stopTime. When it stops in GUI mode, it requires the user to press Start in the control panel to go on.

+c <checkTime> takes an integer argument specifying the time-step at which the program will start to search for a cyclic behaviour (checkTime). At checkTime, if there is a cyclic behaviour and the level of verbosity (set after +v) equals or exceeds the verbosity level required for *cyclicBehaviour* (set in the 'verbosity file'), the cycle will be detected. Stable universal cooperation and stable universal defection are detected always as soon as they occur, as long as the verbosity level equals or exceeds the verbosity level required for *Equilibria* (even if the +c option is not set). If the simulation has not reached a cycle at checkTime (either because checkTime is too low or because the initial conditions do not lead to a cyclic behaviour), the program issues a warning as soon as it detects that condition and stops the simulation. Therefore, in order to detect cycles, the argument of +c must not be very low; otherwise, the cycle will not be detected. If the simulation has reached a cycle at checkTime, it will be detected. It is important to note that in the process of detecting any kind of cycle there is the implicit (not-necessarily-true) assumption that the cycle is sustainable, *i.e.* that Agents can survive indefinitely going through that cycle repeated times. If that is not the case, Agents will eventually die and the cycle will be broken.

+d <decisionsString> is to do deterministic runs, as opposed to runs with stochastic components. +d takes a string argument such as CCDDCDDCD. Each letter represents a decision (Cooperate or Defect). When, according to the program requirement specifications, a random decision should take place, if this option is used, the 'random' decision will be taken from the string argument from left to right. As an example, if the string argument is CDD, the first 'random' decision will be Cooperate, the second one Defect, and the third one Defect. Note that this is for the model, not for a specific Agent. If the string is too short, the program aborts, but it kindly lets you know what length would be sufficient, according to your parameters[2].

+b makes graphs in black & white.

parameterFile is the name of the main parameter file. It is the only required argument.

-s tells the program to use the current time to determine the seed value to use.

[2] Please, note that the sufficient length provided is only valid for games in which there are no players who get the same payoff when they select different actions (otherwise the number of random decisions could be infinite), where Agents do not die (otherwise the number of random decisions could be infinite), and where the forward Memory is equal to 1 (otherwise the number could be infinite or so high that, although we could still give an upper bound, it would not be of much use because exploring the parameter space would be intractable).

-S <seed> takes a 32-bit integer argument specifying the seed. This option is not available in Swarm 2.1.1.

-b sets batch mode run rather than GUI mode run, which is the default mode run. Batch mode is faster than GUI mode, and suitable for running as a background process.

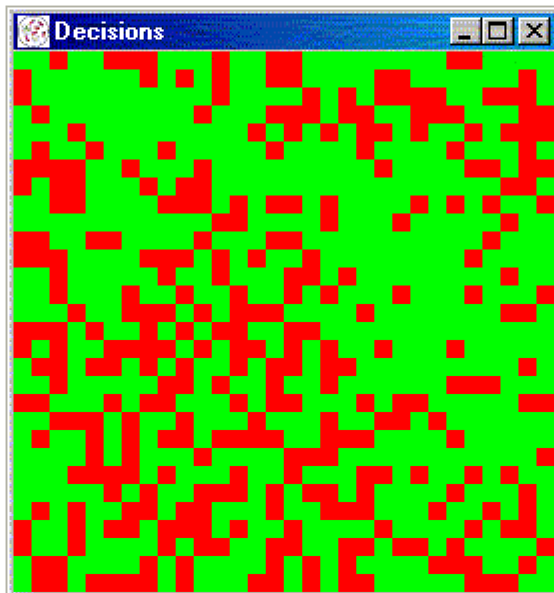
```
Examples:
../../casd +v 1 +V ../../casd.verby +c 50 1x2.scm -S 1239574632
../../casd +v 1 +V ../../casd.verby +c 50 +t 100 +d CCCCDDCC 1x2.scm -b
```

3.3.3. Graphical User Interface (GUI) Mode

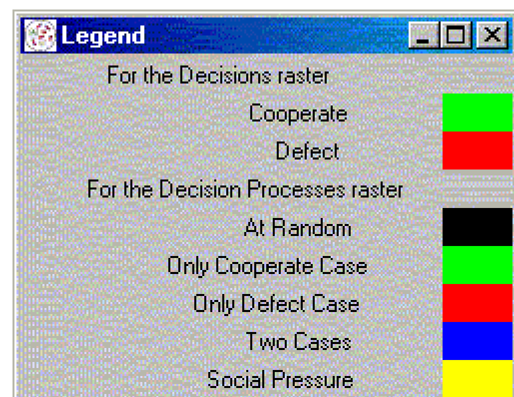
GUI mode is typically used for exploratory runs of the model and for generating pictures for diagrams. When the model is first started in GUI mode, three windows appear. You can then change the value of the parameters for the model and for the observer (do not forget to press return after each entry!) before pressing 'Next' or 'Start' in the 'ProcCtrl' window to begin running the model. This way of modifying parameters is not very recommended, as no record of the parameters will be saved. You cannot change the parameters once the run has started.

Another point worth mentioning about GUI mode is that the stopTime parameter behaves differently in GUI mode than it does in batch mode. In batch mode, stopTime is necessary to specify when the simulation should terminate. In GUI mode, however, this is not required, as the 'Quit' button is there for you to exit at any time. If stopTime is specified, then the simulation will pause at stopTime. You can then click 'Next' or 'Start' to keep running, or 'Quit' to exit. The displays implemented in model 0 are the following:

Decisions Raster & Legend

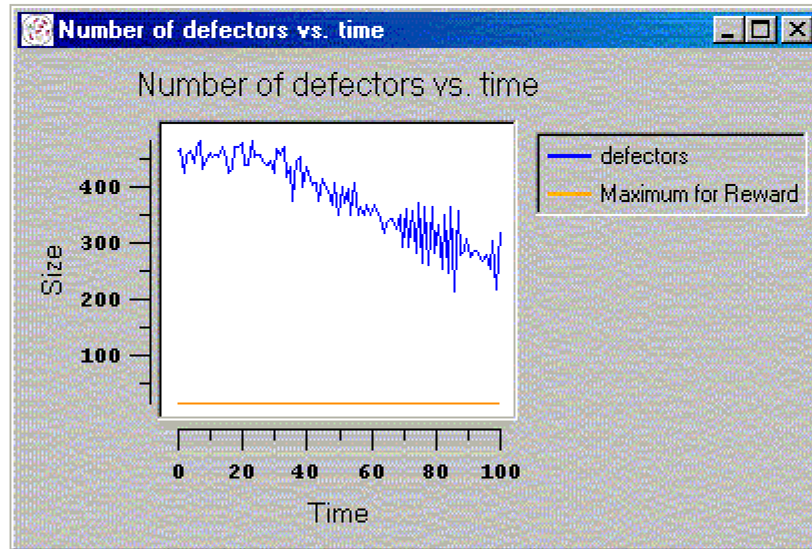


The decisions raster shows the decision that every agent is making in a certain time-step. Cooperators are drawn in green and defectors are drawn in red. The legend shows the colour keys for the rasters.

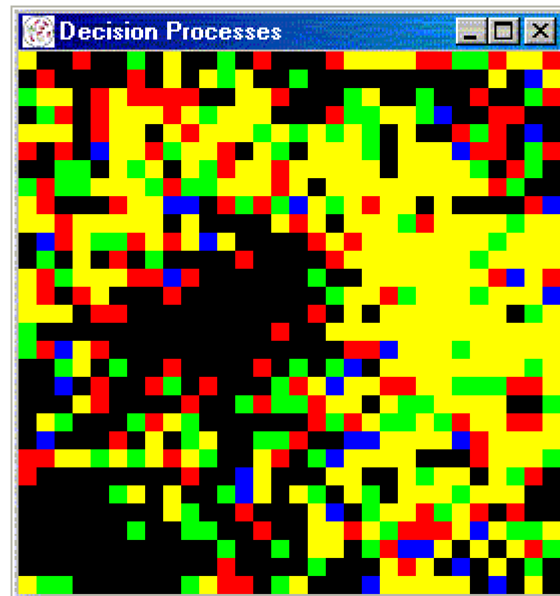


Defectors time-series

The defectors time-series graph reflects the historical state of the decisions raster. It also displays a line showing the maximum number of defectors for which the reward will be given. This is a number that is declared in the payoff Calculator class. For example, in the Symmetric2x2 game structure, the maximum number of defectors is 0, and in the Tragedy of the Commons game the number is entered by the user.

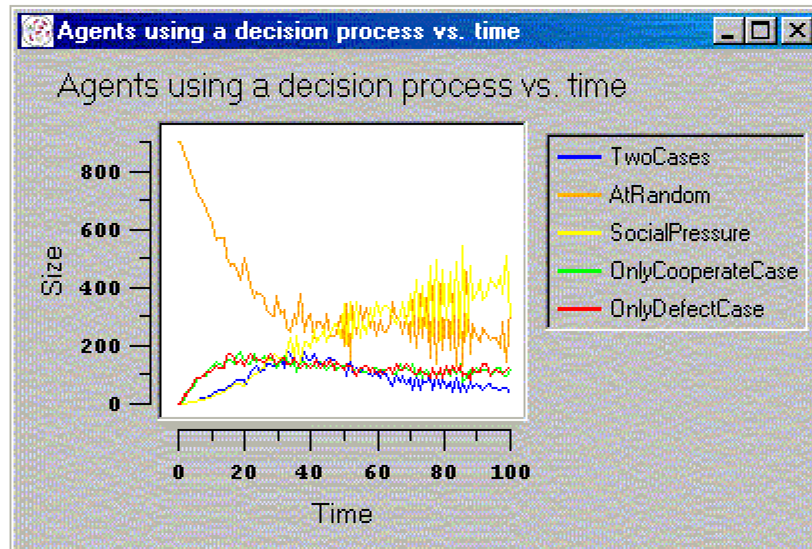


Decision Processes Raster

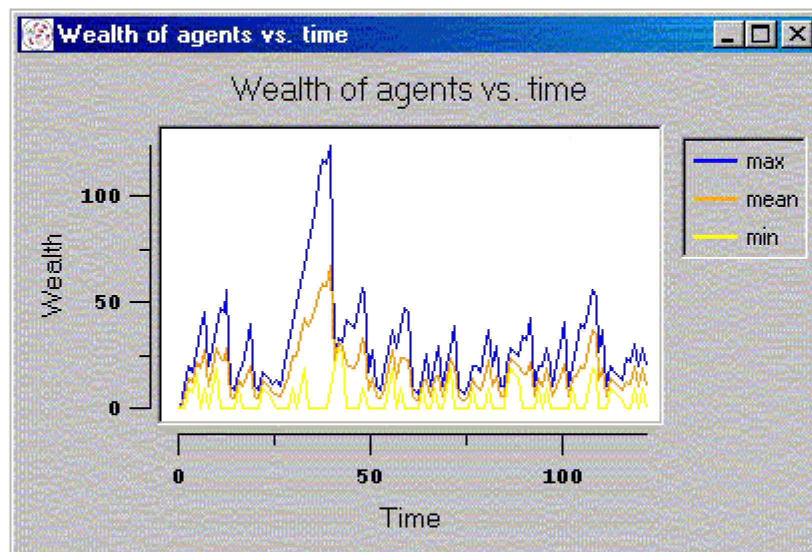


The decision processes raster shows the type of decision process that every Agent has employed in a certain time-step. There are five types of decision process: Random (black), having only a case for a similar situation where the Agent cooperated (green), having only a case for a similar situation where the Agent defected (red), having at least one appropriate case for each decision (blue), and deciding by social pressure (yellow). An Agent decides by social pressure when they would defect if they considered only payoffs, but the last time they defected in a similar situation the average number of disapproving neighbours exceeded their Peer Pressure Threshold. Yellow supersedes the other colours. For example, if an Agent which has at least one appropriate case for each decision chooses by social pressure, then it will be drawn in yellow. Black supersedes blue.

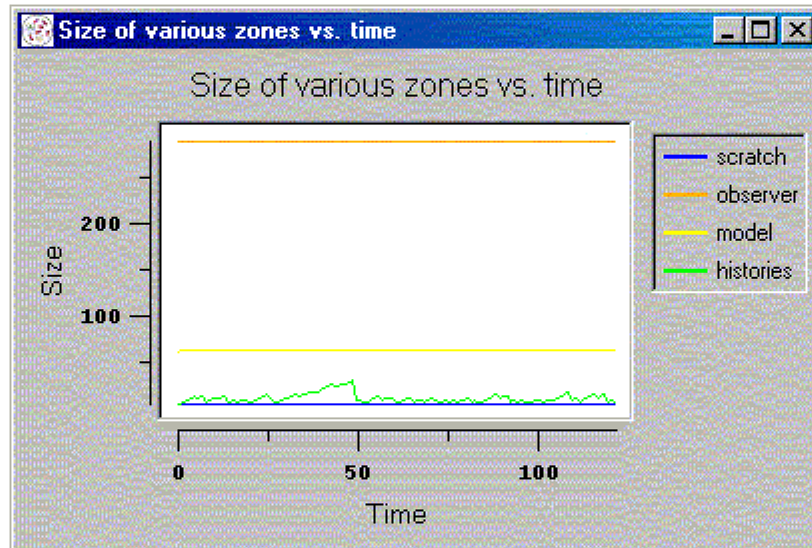
Decision Processes time-series
The decision processes time-series graph reflects the historical state of the decision processes raster.



Wealth time-series graph
The wealth time-series graph shows the maximum, minimum and average wealth in every time step.



Memory zones time-series graph
This graph shows the number of items in various Swarm Zones each time step (useful for tracing memory leaks).



3.3.4. Batch mode

As mentioned in section 3.3.2, to run CASD in batch mode, give the -b flag to the command line. A typical command to run the model in batch mode would look like this:

```
/full/path/to/casd +v 1 +V /full/path/to/casd.verby +c 50 +t 100
1x2.scm -b -s
```

One point to note from this example is that the model is being run from the directory containing all the parameter files. If this is not done, then the parameter filename should include the full path. In batch mode, the +t option is required to specify when the simulation should terminate.

3.3.5. Analysing thousands of runs: Scripts

To analyse how the system *usually* behaves, it is necessary to run multiple simulation runs with different random seeds and analyse their output. To do this, Gary implemented an extremely useful perl script that we provide in the release of CASD-0. The name of the script is cbr-replicator.pl, and it can be found in the "script" directory.

In the usage synopsis below, curly brackets { } are used to denote one of a series of options separated by a vertical bar | for an element value, square brackets [ ] are used to denote an optional element, and angle brackets < > to describe some value the user should provide.

```
./cbr-replicator.pl <mode> [{<runs>|<filename>}] <prog> <param>
[<cmd params...>]
```

The command-line options are explained below:

<mode> can be either 'Random' or 'Deterministic'. In Random mode, multiple runs will be conducted either using the –s flag (use the current time to determine the seed value) or using a file with our own seeds. In Deterministic mode the –d flag will be used when running CASD. The script will work out the maximum number of random decisions that can occur in a run given the parameters provided (note restrictions explained in footnote 2!), and run as many simulations as necessary (using different deterministic strings) to exhaustively explore the whole range of possible outcomes for the given parameters.

<runs> applies only if mode is 'Random'. It specifies the number of runs that will be conducted using the –s flag.

<filename> applies only if mode is 'Random'. It specifies a file with seeds to use. This option will not work in Swarm 2.1.1, where the –S flag is not available.

<prog> is the full path to the program to use (casd).

<param> is the main parameter file to use with the program.

<cmd param...> are the command line parameters to send to the program. These should be with all + (model specific) options first, - (Swarm specific) options last.

Examples:

```
../../script/cbr-replicator.pl Random 10 ../../casd 1x2.scm +v 1 +V
      ../../casd.verby +c 50 +t 70 -b

../../script/cbr-replicator.pl Random cbr-rep-20040309-130032.seeds
      ../../casd 1x2.scm +v 1 +V ../../casd.verby +c 50 +t 70 -b

../../script/cbr-replicator.pl Deterministic ../../casd 1x2.scm +v 1 +V
      ../../casd.verby +c 50 +t 70 -b
```

The outputs of the script are three files with the following names:

cbr-rep-DATE-TIME-results.txt (*e.g.* cbr-rep-20040309-130032-results.txt)

This file presents a count of the number of simulation runs that have ended up in a cycle of length *x* where the reward has been given *y* times.

```
Stable Cooperations:    3 times
Stable Defections:      5 times
Reward given      0     out of      1      times:     5
Reward given      1     out of      1      times:     3
Reward given      1     out of      2      times:     2
Total:      10
```

The reward rate of a cycle equals y/x. The file provides the average reward rate and its standard error over all the cycles.

```
Average reward rate:    0.4
Standard Error of the average reward rate:      0.145296631451356
```

Finally, it presents the average payoff obtained by the group and its standard error, calculated over all the cycles.

```
Total:      10
Average Group Payoff:   9.6
Standard Error of the average Group Payoff:     2.03415284031898
```

cbr-rep-DATE-TIME.log (*e.g.* cbr-rep-20040309-130032.log)

This is a log file where the progress of the actions performed by cbr-replicator is saved.

cbr-rep-DATE-TIME.seeds (*e.g.* cbr-rep-20040309-130032.seeds)

This is a file where all the seeds used by cbr-replicator are saved. It is produced only in Random mode when the user requires conducting a certain number of runs.

3.4. How to extend the program: Implement your own games

To implement new games, you will have to implement a new class which must follow the PayoffCalculator.h protocol and be subclassed from SwarmObject. Following the PayoffCalculator.h protocol basically means that the new class must implement the following methods:

```
-initialiseWithModel: (ModelSwarm *)m;
-(void)updateDecisions;
-(DoubleSimple *)getPayoffForDecision: (decision_t) decision;
-(DoubleSimple *)getPayoffForFullCooperation;
-(DoubleSimple *)getPayoffForFullDefection;
-(BOOL)isRewardGiven;
-(int)getMaximumDefectorsForReward;
```

You can find examples in the classes: Symmetric2x2 and Reward. Both classes are extensively commented so if you want to implement a new game the best strategy is to read the header file and the implementation file of either of those two classes.

Once you have implemented your game class (YourGameClass.h and YourGameClass.m), you will have to add its name to the Makefile so it gets compiled and linked with the other classes in the simulation. Specifically, append YourGameClass.o after the other Payoff classes:

PAYOFFS= Symmetric2x2.o Reward.o **YourGameClass.o**

Compile the program again and then you can use your new game. Remember to write the name of your class and the name of your parameter file in the main parameter file:

```
(list
 (cons 'modelSwarm
       (make-instance 'ModelSwarm
                      #:bMemory 1
                      #:fMemory 1
                      #:descriptorOtherDefectorsStr YES
                      #:descriptorMyDecisionsStr YES
                      #:expThresholdStr DoubleSimple=20.0
                      #:peerPressureThreshold 1
                      #:envShape planar
                      #:nbrhood moore
                      #:xSize 1
                      #:ySize 2
                      #:radius 1
                      #:payoffCalClassStr YourGameClass
                      #:payoffParameterFile
                           YourGameClassParameterFile.scm)))
```

YourGameClassParameterFile.scm will probably look something like this:

```
(list
 (cons 'payoffCalculator
       (make-instance 'YourGameClass
               #:aPayoffString DoubleSimple=10.0
               #:anotherPayoffString DoubleSimple=9.0
               #:oneMorePayoffString DoubleSimple=2.0
               #:oneInteger 15)))
```