# Manual to CASim

## JAVA-Environment for Simulating Cellular Automata

**Uwe Freiwald and Jörg Weimar**
TU Braunschweig, 1999/2000

The simulation system CASim is an environment for simulating cellular automata. In this manual, the usage of the simulation environment is described. The system CASim is written entirely in Java and consists of three different programs:

- a programm with graphical user interface *CASimFrame*
- a JAVA Applet with graphical user interface *CASimApplet*
- a program without graphical user interface (mainly for performance measurements) *CASim*

The code is tructured in two JAVA packages *de.tubs.cs.sc.casim* and *de.tubs.cs.sc.cavis*. For the programs with GUI, both packages are necessary, for the command-line interface only *casim* is needed. You need to ensure that either the archive `casim.jar`, which contains all necessry classes from both packages, or the directories `casim` and `cavis` are included in your CLASSPATH.

# 1 The simulation environment CASimFrame

The program CASimFrame presents a graphical user-interface for the specification and simulation of cellular automata. The exact appearance depends on the operating system and the Java virtual machine. The texts of the user-interface are either in German or in English, depending on the language setting of your computer. CASimFrame implements a single document interface, i.e.g, only one cellular automaton is in use at any one time. You can simulate several automata at the same time by repeatedly starting CASimFrame, or you can couple different cellular automata using the program CAComb from a different project.

## 1.1 Starting CASimFrame

You can start CASimFrame like any other JAVA-program from the command line of your operating system with the following command:

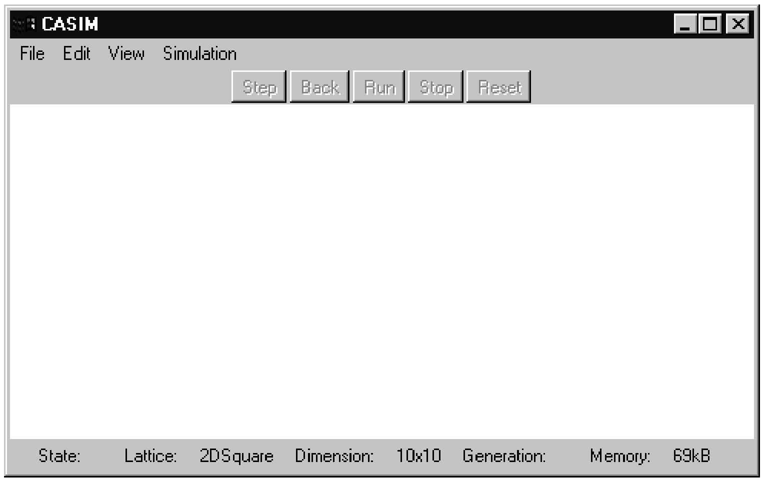

Figure 1: CASimFrame running under Windows

```
java de.tubs.cs.sc.cavis.CASimFrame
```

On Unix-systems you could use

```
java de.tubs.cs.sc.cavis.CASimFrame &
```

to start the program in the background. After a short setup-time you should see a window similar to the one in Figure 1.

For simplification, the archive `casim.jar` contains a small class `casim` so that the program can be started simply with

```
java casim
```

You may also add one or several files on the command line, which are then loaded as in the menu option "File-Open".

On all systems additional output is printed to the shell, most error messages appear as small dialog boxes.

By default, CASim starts with a two-dimensional lattice of 10 by 10 cells, so in the beginning you don't need to specify any lattice settings.

In the next step, you need to decide how to describe the cell state and the state transition function. The simulation system CASim supports three ways of doing this:

- You can specify the state and state transition function using an interactive dialog.
- You can program a Java class, in which only a few methods must be implemented, so a very basic knowledge of Java is sufficient.


Figure 2: Create a new CA with File - New...

- You can specify a cellular automaton in the special-purpose language CDL, which is then translated automatically into Java.

The first possibility is recommended for non-programmers. In this case the state of the CA is restricted to a subset of the natural numbers, which is sufficient for most cases.

The second prossibility allows far more freedom in describing the state and state transition function and is described in section 3. The third possibility of using CDL is described elsewhere.

# 2 Graphical specification

In this section, step-by-step instructions are given on how to specify and simulate a cellular automaton without writing any program code.

## 2.1 State transition function specified interactively

**A new cellular automaton** Create a new cellular automaton using the menu item *File - New...* . You will see a dialog window in which you can enter the complete state definition.

### 2.1.1 Name and Number of States

Now enter the name of you cellular automaton at the left upper corner. On the right, enter the number of states. States are numbered $0 \cdots max - 1$, and you should enter the number of states before starting to specify the transition rule or the color table.

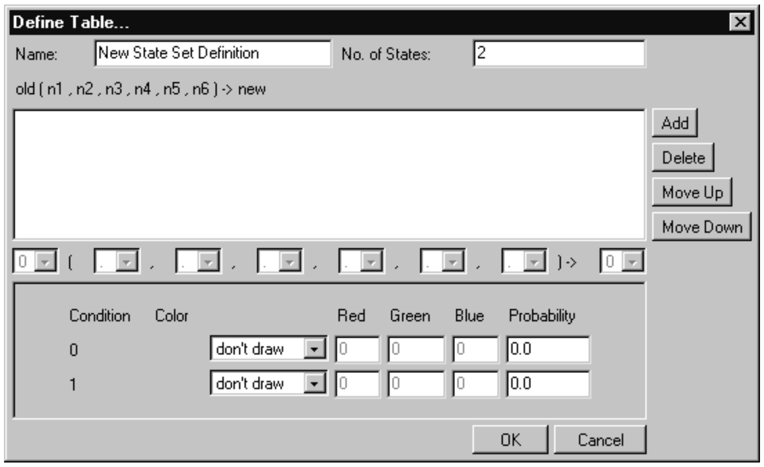


Figure 3: Dialog for defining the state set, transition function, and presentation.

After entering the name and number of states, you see in the central part of the window a list of state transition rules, and in the lower part a table with one row per state.

### 2.1.2 State transition rules

You can now enter the rules for the state transition. The default is that the state remains unchanged.

**A new rule** To add a new rule, click on *Add*. This will add a new rule with default settings to the list of rules.

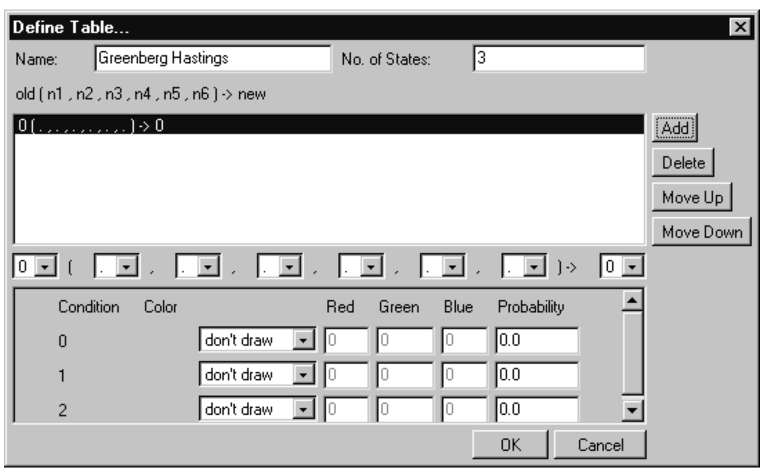


Figure 4: A new state transition rule

Every rule consists of three parts. On the left, you enter the current state of the cell by selecting it from the list of possible states. In the middle, you specify the state of the neighbors, and on the right the new state for the cell.

As an example, to specify the rules for the Greenberg-Hastings CA with the transition rules

$$\begin{array}{lcll} 0 & \rightarrow & 2 & \text{if at least one neighbor is 2} \\ 0 & \rightarrow & 0 & \text{otherwise} \\ 2 & \rightarrow & 1 & \\ 1 & \rightarrow & 0 & \end{array}$$

enter a name, and 3 for the number of states. Then add a new rule, select 0 as the current state, 2 for the first neighbor, an 2 for the new state, and thus the first rule is entered. The following rules do not need any neighbor information. The simulation system checks the rules for a match from top to bottom. Figure 5 shows all four rules.


Figure 5: Four state transition rules for the GH-CA.

In this simple mode the geometry of the neighborhood is not taken into account. The system simply determines whether there is one neighbor in each of the neighborhood states given in the state transition rule, without regard to the order. In the example, the first rule simply says that at least one neighbor must be in state 2. Neighbors with the value "." are wildcards and can have any value.

**Moving rules** In the example there are two rules for the currrent state 0. The first one matches the situation where at least one neighbor has value 2, the second one matches any situation. Therefore the order of these rules is important, since they are evaluated from top to bottom. You can change the order of the rules with the buttons *Move up* and *Move down* .

**Deleting rules** You can delete a rule by selecting it and clicking on *Delete.*

Now you have determined the state transition function, and now need to determine how states should be shown graphically.

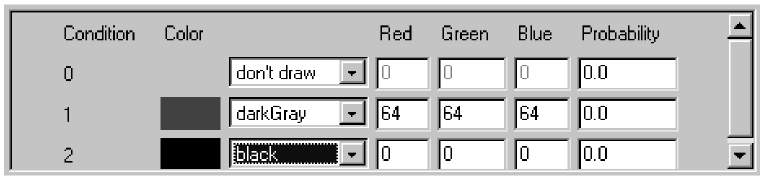


Figure 6: Colors for the states

### 2.1.3 Table of states

The lower part of the dialog window shows a table of all states. In this table you can specify which color should be used to present cells with this state. Normally you should use colors which are easily distinguishable. For three-dimensional CA, different sides of a cubic cell are shown in different brightness, so use color to distinguish different states.

Figure 6 shows an example. Every row shows the properties of one state, from left to right:

- The number of the state
- The color in which it is shown
- A selection of standard colors
- The red part between 0 and 255
- The green part between 0 and 255
- The blue part between 0 and 255
- The initial probability of this state.

**Color** You can use the full scale of colors for representing a state. A color can be specified by name, or by a triplet of RGB values, i.e., red, green, and blue values in the range 0 to 255. White is the triplet (255,255,255), black is (0,0,0), pure red is (255,0,0) and so on. Sometimes it is useful not to show certain cells, this is achived by selecting *don't draw*. Especially in three-dimensions this possibility allows the user to see cells in the interior of the cube (at a considerable performance price, so if you prefer faster representation of only the outside cells, select a color for each state).

**Initial probabilities** CASimFrame can initialize the lattice with different states according to given probabilities. Enter probabilities between 0.0 and 1.0. The system will try to ensure that the sum of probabilities remains 1.0 by adjusting

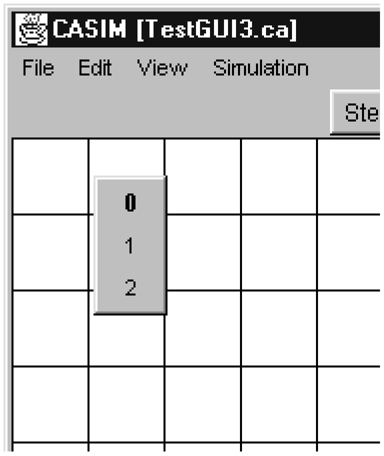


Figure 7: Menu for initializing a cell

the probability for the state 0. CASimFrame then initializes the cells randomly with the selected probabilities.

Confirm all entries by pressing *OK*. The lattice should now be initialized.

## 2.2 Initializing selected cells

As a last step before simulation you can select an initial state for each individual cell. Turn on display of the grid so that you can identify the cells. Then by clicking on a cell with the right mouse button, you pull up a menu which lets you select a new state for this cell. The current state of the cell is highlighted. Examples are shown in Figure 7.

Continue changing cells until you have reached the desired initial conditions. You may also use this possibility during the simulation to modify individual cells. The menu item *Simulation - Reset* resets the lattice to new probabilistic initial conditions.

This concludes the description of how to specify a cellular automaton using the graphical user interface. You can skip the description of how to specify a CA in Java and go directly to the simulation in section 4 on page 17.

# 3 Transition function in JAVA

The central purpose of CASim is to simulate cellular automata. A cellular automaton is described in CASim by an object derived from the class `State` in the package `de.tubs.cs.sc.casim`, which represents the interface to the user-specified CA. To program a new cellular automaton, only basic knowledge of JAVA is necessary.

As a first step, it is best to let the simulation system create skeleton code into which you then insert your own code. This skeleton code is created with *File - New JAVA File* . You will be asked for a file name, and you should use the name of you CA (no spaces) and the ending ".java". The resulting file looks like the one in Figure 8.

Please note that in Java, a class must be declared in a file with the same name as the class name and that this name must begin with a letter. Upper- and lowercase letters are distinguished even on operating systems like Windows.

This newly generated Java file can now be edited to include the code specific to the cellular automaton to be described. The file conatins a number of comments which indicate the positions where changes need to be made. in Java, comments start with //, /* oder /**. Comments starting with // end with the end of the line, the other comments must be ended explicitly with a closing */. Nested coments are not allowed. The skeleton file contains numbered comments, and we refer to these numbers in the following sections.

## 3.1 Declaring the cell state, step (1)

JAVA allows you to use the full power of a general programming language for specifying the cell state. You can use one or more variables to specify the state of a cell. You can use the primitive JAVA data types

- int, short, long, byte: integer numbers with different ranges
- float, double: floating-point numbers
- boolean: a boolean value (true or false)
- char: a character

or you can use more complex types through the definition of special (inner) classes. This option is not usually necessary for cellular automata.

For the example Greenberg-Hastings CA, a simple integer is sufficient. JAVA does not have enumeration types or types to declare a subrange of an integer.

```
private int condition;
```

Here `private` means that the variable `condition` may only be used inside the class.

```
/**
 * GreenbergHastings.java
 * generated by CASIM
 */

import de.tubs.cs.sc.casim.State;
import de.tubs.cs.sc.casim.Cell;
import de.tubs.cs.sc.casim.Lattice;
import java.awt.Color;

/**
 * GreenbergHastings
 */
public class GreenbergHastings extends State {
    /** (1) add state variables here
     * e.g. int condition;
     */
    /** (2) add initialization of state variables here
     */
    public void GreenbergHastings() {
    }
    /** (3) add lattice initialization here
     */
    public static void initialize(Lattice l, int option) {
    }
    /** (4) add drawing color definition here
     */
    public Color getColor() {
      return null;
    }
    /** (5) add state variables copy code here
     *      do a field to field copy from s to this
     */
    public void copy(State s) {
      GreenbergHastings source = (GreenbergHastings)s;
    }
    /** (6) add transion function code here
     */
    public void  transition(Cell cell) {
    }
    /** (7) add code for definition of constant states here
     */
    public State getConstant() {
      return null;
    }
}
```

Figure 8: JAVA skeleton code

## 3.2 Initialising the state, step (2)

To make sure that each cell is in a defined state, all components of the state should be initialized with a default value. A good place to do this is in the contructor, which is called when a new object is created.

For example, one could insert the follwing code in the constructor:

```
condition = 0;
```

## 3.3 Initialising the lattice, step (3)

For most experiments with cellular automata, the lattice should be initialized to a non-trivial state. This is done in the static method `initialize`. The method initialize can access the lattice geometry and all cells through the object `l` of class `Lattice`. A second parameter `option` is used to distinguish between different experiments. This is an integer which can be set from the user interface at simulation time.

The class `Lattice` provides information about the size of the lattice with the methods `int getX()`, `getY()`, and `getZ()` and access to the individual cells through the method `State getState(...)`, which takes the coordinates as arguments. Note that the objet returned by `getState` must be typecast to the class you are programming before the state variables can be accessed. In the example, a simple initialization could be

```
((GreenbergHastings)l.getState(3,3)).condition = 2;
```

Note that cell numbering starts with 0. This example only works for lattices of the correct dimension and size. A more complex initialization which works for any lattice size and dimension is:

```
int x = l.getX()/2;
int y = l.getY()/2;
int z = l.getZ()/2;

((GreenbergHastings)l.getState(x,y,z)).condition = 2;

if (option == 1){
    ((GreenbergHastings)l.getState(x+1,y,z)).condition = 1;
}
```

## 3.4 Color, Step (4)

Each cell can be graphically represented by a colored patch, a textual representation, or an icon. To use text, add a method

```
public String toString()
```

to the code which returns a string representing the state.

To use color, insert code into the `getColor`-method to return a color value depending on the state. One possibility is to use predefined colors and select one color with a `switch` statement:

```
switch(condition) {
    case 1: return Color.darkGrey;
    case 2: return Color.black;
}
return null;
```

In this case state 0 will not be colored, since `getColor` returns `null` in this case, which is also the default setting.

Fo rmore complex automata, and if colors different from the predefined colors are to be used, it is convenient to create an array of colors which are then returned depending on the state:

```
private static Color[] colors;
static {
    colors = new Color[3];
    colors[0] = null;
    colors[1] = new Color(150,150,255); // pink
    colors[2] = Color.red;
}

public Color getColor() {
    return colors[condition];
}
```

Colors can also be created on the fly depending on the cell state, such as in

```
public Color getColor() {
    return new Color(50*condition,50*condition,50*condition);
}
```

### 3.5 Icons

To display the state of a cell using a small icon instead of a colored field, one can implement the method

```
public java.awt.image.ImageProducer getIcon();
```

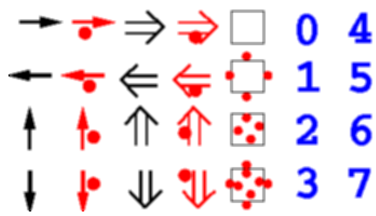


Figure 9: Image with icons for a celluar automaton

which tells the simulation system which image to display for this cell. This method is only called if `getColor()` returned `null`.

To simplify the use of icons, a helper class `de.tubs.cs.sc.cavis.IconManager` is included in the simulation system. This class makes it posible to create one image containing many or all needed icons, and then to select one icon in the image depending on the state of a cell. The iconmanager can be called as a program to create code that can be included in the new class. In this case, three parameters must be given, first name of the image file, which may be a GIF file or a JPEG file. Then, two parameters give the number of icons contained in the image in x- and in y-direction. An example is:

```
java de.tubs.cs.sc.cavis.IconManager arrvn.gif 7 4
```

for the image 9 in the file “arrvn.gif” with $7 \times 4$ icons.

This call creates code that contains the complete information from the image file, so that the image file need not be present any further in the simulation. The code should be included in the new class.

To display a cell as an icon, the method `getIcon()` uses the IconManager to display the icon from a certain position in the given image:

```
public java.awt.image.ImageProducer getIcon(){
    if (...) return im.getIcon("arrvn.gif",state%7,state/7);
};
```

## 3.6 Copy-method, step (5)

Due to the synchronous updating, the simulation system must store both the $n$th and the $n+1$st generation of states. Therefore the states must be copied after each generation. Since the simulation system does not know how the state is stored, the user must supply a method `copy` which copies the state from another object. This method should simply copy all state variables, in the example:

```
public void copy(State s) {
    GreenbergHastings source = (GreenbergHastings)s;
```

```
    condition = source.condition;
}
```

In the second line, the parameter s is type cast into the new Java class, then all state variables (here only one) are copied.

## 3.7 State transition function, step (6)

The transition from one generation to the next is specified in the method `transition`. Here you can use the full power of the programming language Java to express the state transition function, such as loops, `if`-statements, or arrays for storing tables.

The method `transition` has one parameter `Cell c`. Through thi sparameter, access to the neighbors is provided. The method `c.getNeighbors()` returns an array of `State`s, which conatins all neighbors of the current cell. The cell itself is included in this neighborhood, and is always the first element in the array. To make the code efficient, the method `c.getNeighbors()` should only be called when necessary, i.e., for those states where the transition function really does depend on the neighbors.

The example Greenberg-Hastings CA could be programmed as shown in Figure 10. In this example, only if `condition` is zero is it necessary to consider the neighbors. Therefore a call to `c.getNeighbors()` is used to obtain an array of `State`s. Then a loop over all elements of this array is used to determine if at least one neighbor has value 2. Note that the elements need to be type cast into objects of the current class (in this case `GreenbergHastings`) for the state variables to be accessible. This leads to the expression `((GreenbergHastings)neighbors[i])`. If no neighbor has value 2, the variable condition remains unchanged, in this case with value 0.

With these changes, the celllular automaton has been programmed completely in Java. Figure 11 shows the complete sourcecode which must now be compiled and loaded in CASimFrame.

## 3.8 Loading a JAVA class in CASimFrame

After the file has been modified to describe the cellular automaton the user wishes to simulate, this file needs to be compiled and loaded in the simulation environment CASimFrame. If the Java Development Kit is installed, the system can compile and load the java file automatically. In some cases it may be necessary that the file `tools.jar`, which can be found in one of the library folders of the Java installation, be included in the environment variable `CLASSPATH`. To load a java file, simply select the menu item *File - Open* and select the Java file for opening. This file will then be compiled and loaded, You can also compile it separately, and then load the `.class` file.

```
public void  transition(Cell cell) {
    if (condition>0) {
        condition = condition-1;
    } else {
       State neighbors[] = cell.getNeighbors();
       for (int i=0;i<neighbors.length;i++) {
          if (((GreenbergHastings)neighbors[i]).condition==2) {
              condition = 2;
              break;
          }
       }
    }
}
```

Figure 10: State transition function in JAVA

If the compilation was not successful, you will see a dialog window, and should compile the file separately to see the compilation error produced. Otherwise you will see the initial state of the CA as in Figure 12.

The simulation of the CA proceeds in the same way for states specified with Java as for states specified interactively, and is described in the following section.

Loading a class can also be achieved from the command line by specifying a file with the ending `.html`, which describes a configuration, or a file with ending `.java` or `.class`, which conatins only the class to be loaded and not the lattice configuration. It is also possible to first specify a `.java` file, which is then translated, and as a second argument a `.html` file, which specifies the lattice configuration (for example `java casim MyClass.java MyClass.html` ).

```java
/**
 * GreenbergHastings.java
 * generated by CASIM
 *
 */
import de.tubs.cs.sc.casim.*;
import java.awt.Color;

public class GreenbergHastings extends State {
    /**
     * (1) add state variables here
     */
    private int condition;
    /**
     * (2) add initialization of state variables here
     */
    public void GreenbergHastings() {
        condition = 0;
    }
    /**
     * (3) add lattice initialization here
     */
    public void initialize(Lattice l) {
        ((GreenbergHastings)l.getState(3,3)).condition = 2;
    }
    /**
     * (4) add drawing color definition here
     * returning null means the cell will not be drawn
     */
    public Color getColor() {
        switch(condition){
            case 1:  return Color.darkGrey;
            case 2:  return Color.black;
        }
        return null;
    }
    /**
     * (5) add state variables copy code here
     */
    public void copy(State s) {
        GreenbergHastings source = (GreenbergHastings)s;
        condition = source.condition;
    }
    /**
     * (6) add transion function code here
     * evaluating the new state of the cell
     */
    public void  transition(Cell cell) {
        if (condition>0) {
            condition = condition-1;
        } else {
            State neighbors[] = cell.getNeighbors();
            for (int i=0;i<neighbors.length;i++) {
                if (((GreenbergHastings)neighbors[i]).condition==2) {
                    condition = 2;
                    break;
                }
            }
        }
    }
}
```

Figure 11: JAVA sourcecode of the Greenberg-Hastings CA

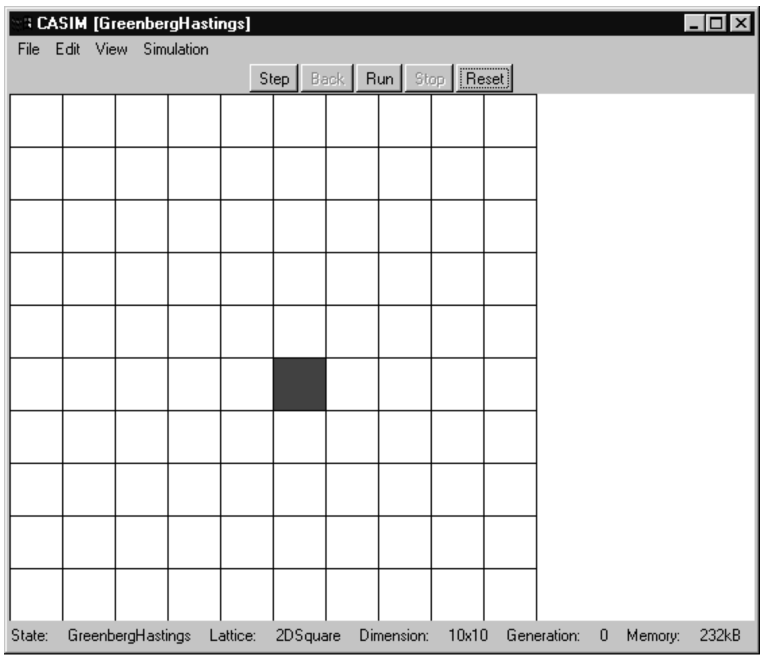


Figure 12: Greenberg Hastings CA in the initial state.

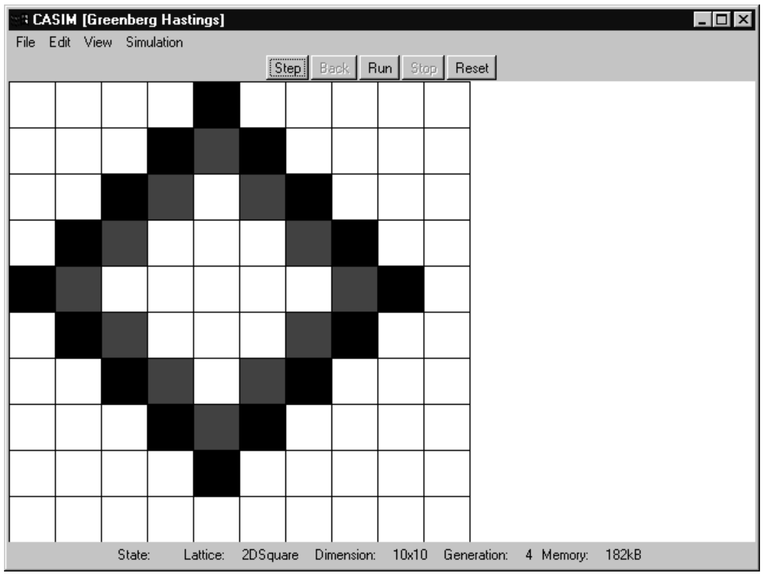


Figure 13: Greenberg-Hastings after 4 steps

# 4 Simulating the CA

The simulation of the cellular automaton is controlled by the buttons on at the top of the simulation window. These buttons are labeled *Step, Back, Run, Stop* and *Reset*.

First, click on *Step*. The program CASimFrame now simulates the first time step. After executing three more steps, your automaton should look like the one in Figure 13. This concludes the description of the basic steps needed to simulate a CA. All further controls are described in the following section.

## 4.1 Usage of CASimFrame

## 4.2 The menu

The menu of CASim has the following entries:

- File
  - New
  - New JAVA File
  - Open
  - Save
  - Save As
  - Print

  - Exit

- Edit
  - Lattice
  - Direct Specification

- View
  - Grid
  - Coordinates
  - Text
  - Statistics
  - Pair Statistics
  - Clip Corner
  - History 1D
  - show more border
  - show less border

- Simulation
  - Step
  - Back
  - Run
  - Stop
  - Reset
  - Steps:

The individual menu entries are explained below.

#### 4.2.1 The menu File

**New** The menu item *File - New* creates a new cellular automaton for interactive specification. A dialog appears as described in Section 2.1.

**New JAVA File** The menu item *File - New JAVA File* allows you to create a new JAVA file which can then be filled with information as described in section 3. First a dialog appears, in which a file name should be selected or entered with the ending `.java`.

**Open** CASimFrame can operate on five different types of files:

- JAVA sourcecode files (ending in `.java`)
- JAVA bytecode files (ending in `.class`)
- CDL files (ending in `.cdl`)
- Files that contain the interactively specified state transition and have been produced by CASimFrame (ending in `.ca`)
- Files that describe a complete simulation, including lattice parameters and visualization settings (ending in `.html`).

When you call *File - Open* , you are asked to enter a file name or select a file. The ending of the file name decides which type of file is read, and what the simulation system does with the content.

JAVA source or bytecode files should always contain the description of a cellular automaton, i.e., a class derived from the class `State`. Source files are compiled and then the class is loaded. If loading the class was successfull, the simulation system directly initializes the cellular automaton.

**Save** With *File - Save* you can save the currently active cellular automaton if it was specified interactively. The resulting file should have ending `.ca` and allows you to load the specification again for modification or presentation. For CA specified from a java program, this option is not available.

**Save As** The menu item *File - Save As* allows you to save many different aspects of the current CA simulation in different types of files. The different types of files are distinguihed by the endings:

.ca If the CA was specified interactively, you can save the specification under a different name.

.java If the CA was specified interactively, you can save the specification as Java code. This Java code contains the complete information from the interactive specification, converted into the form of a Java class derived from the class `TableState`, which in turn is derived from `State`. You can now modify this code and load it as described in section 7.

.html A file in the HTML format is used to store the information on the configuration of the lattice and all the visualization settings which you specify for the simulation. The file also contains a reference to the Java class describing the cellular automaton state, but does not contain this class itself.

The resulting HTML file can be used for WEB-presentations using applets, or to reload the same configuration and view settings using *File - Open*.

.m A file with ending `.m` is used to store the current state of the cellular automaton in a format that can be read by the program Mathematica or by other programs that can read ASCII-text files. The states are described using the result of the `toString`-method.

**Print** *File - Print* allows you to print the current state of the simulation as it appears in the simulatiomn area. First, a dialog depending on the operating system appears which allows you to select a printer, or to print to a file.

The printout contains the currently displayed situation, but with better resolution than on the screen. On most systems you can also print into a file, in which case Postscript is generated. If you want to use the resulting files as EPS-files to be included into other texts, you should use a program to convert them, such as `ps2eps` (for more information contact J.Weimar@tu-bs.de).

**Exit** Select *File - Exit* to leave CASimFrame. You can also leave the simulation system by closing the main window.

### 4.2.2 The Edit menu

**Lattice** The menu item *Edit - Lattice* is used to change the properties of the lattice. It calls up a dialog similar to the one shown in Figure 14. In this dialog

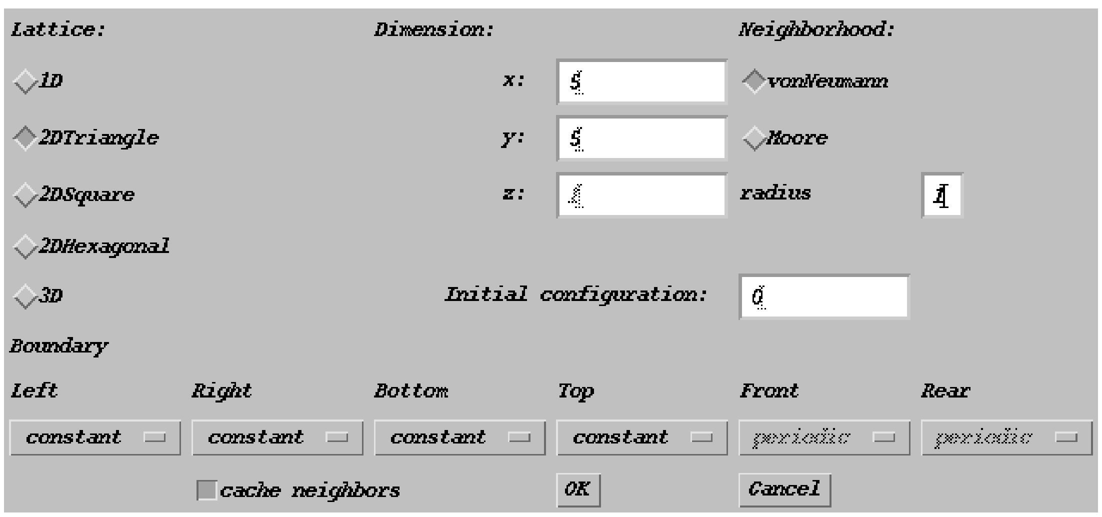


Figure 14: Dialog for selecting the lattice properties

window, all properties of the lattice can be changed. On the left hand side you find checkboxes for selecting the dimension and geometry of the lattice. The system supports lattices of one tpo three dimensions and three kinds of two-dimensional geometries:

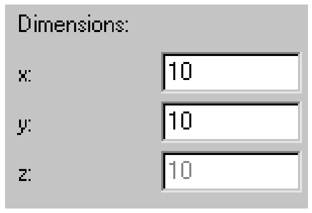


Figure 15: Size of the lattice

- One-dimensional lattice (*1D*)
- Two-dimensional lattice of triangles (*2DTriangle*)
- Two-dimensional lattice of squares (*2DSquare*)
- Two-dimensional lattice of hexagons (*2DHexagonal*)
- three-dimensional (cubic) lattice(*3D*)

Depending on the Dimension of the lattice, you can enter the size in the different dimensions in the center portion of the dialog. x determines the width, y the height, and z the depth of the lattice.

The neighborhood of a cell can be selected as the Moore neighborhood or the vonNeumann neighborhood, both with arbitrary integer radius. In the two-dimensional square lattice, the vonNeumann neighborhood of cell $(x, y)$ with radius $r$ is defined as

$$N_{i,j} = \left\{ (k,l) \in \mathcal{L} \;\middle|\; |k-i| + |l-j| \leq r \right\},$$

i.e., the sum of the distances on the lattice in any dimension between the two cells must be less than or equal to $r$. The Moore neighborhood is defined as

$$N_{i,j} = \left\{ (k,l) \in \mathcal{L} \;\middle|\; |k-i| \leq r \text{ and } |l-j| \leq r \right\},$$

i.e., the largest distance in any of the dimensions must be less than or equal to $r$. These neighborhoods can be generalized to the other geometries, where steps on the lattice are counted in one of the six directions for the hexagonal lattice, and in one of the three directions for the triangular neighborhood. The vonNeumann neighborhood can be straightforwardly adapted to these lattices, while the Moore-neighborhood has only be adapted for the hexagonal case, where the Moore neighborhood of radius $r$ is taken to be those cells, for which the shortest path to the center cell involves a maximum of $r$ steps in any of the six directions. Cells for which a path with not more than $r$ steps in any one direction exists, but is not the shortest path, are ignored.

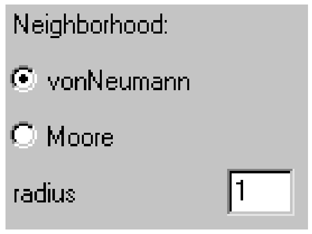


Figure 16: Neighborhoods


Figure 17: Caching the neighborhood

If the problem at hand requires a special neighborhood different from the predefined neighborhoods, you need to use the possibility of defining arbitraty neighborhoods, which exitsts when you specify the automaton in Java or CDL. The settings in the lattice dialog only apply to those cases, where the function `getNeighbors()` is used to obtain the neighbors. In all other cases, such as when the function `getNeighborsMoore()` is used, the setting in the dialog have no effect,

Since the neighborhood does not change from one time step to the next, the simulation system can create a list of neighbors for ach cell and store this list instead of recreating it at every time step. This option is called "caching the neighborhood". The advantage is that execution is faster, the disadvantage is that it needs more memory. The option is on by default, but should be turned off for large simulations where memory is scarce.

For every lattice the boundary conditions need to be specified. The default in CASim is to use periodic boundary conditions. This can be changed for every border of the lattice separately. The possible boundary conditions are periodic, reflective, or constant. Reflective boundary conditions approximate a mirror set at the border of the lattice. Note that for hexagonal and triangular lattices, this is only an approximation in the current implementation. For constant boundary conditions, a constant value is returned for all accesses to cells outside the lattice. The value of this constant can be set by using the constants `constantcell` or `constantcellright` etc. in CDL or by implementing the method `State getConstant()` in the Java code. Different values can be returned for the different boundaries by having a method

```
public State getConstant(int dim, boolean dir)
```

return different values. Here `dim` gives the dimension (x-axis is 1, y-axis 2, and z-axis 3), and `dir` gives the side of the lattice (true for positive end, false for


Figure 18: Boundary conditions

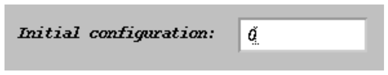


Figure 19: Initialization option

negative end).

To select different initial conditions, the dialog contains a field for entering an integer number. This number is passed as second argument to the method `initialize`. This method in the Java class can then create different initial conditions depending on this value. In theCDL specification, this value can be accesses by the name `option` in the initial section.

**Direct Specification** The menu item *Edit Direct Specification* calls the dialog which allows you to specify a cellular atomaton interactively. This menu item is only active, if the current CA was specified interactively, and not using a Java class. For a detailed description of the interactive specification of CA refer to Section 2.1.

### 4.2.3 The menu View

The menu =*View* provides a number of options for modifying the display of the CA.

**Grid** The cells of the CA can be shown either with a grid separatying the individual cells, or without a grid. Switch betwen these options with *View Grid*. Especially for large three-dimensional CA, or for CA with many (small) cells, the grid should not be shown as it covers too much of the display area. In other cases, showing the grid makes the individual cells more visible. By default, the grid is not shown.

**Coordinates** The coordinates of all cells can be shown by selecting *View Coordinates*This option is not available for three-dimensional lattices, and should only be used with sufficiently small lattices so that the texts for different cells remain separated.

**Text** For each cell, a textual representation can be shown with *View Text*.The textual representation comes from the code in `toString()` in the Java class, or simply the integer code for interactively specified automata. When the CA is specified using CDL, this text is generated automatically.

**Statistics** Many applications require a statistical evaluation of a CA simulation. Such statistics can be displayed using *View - Statistics*. If this option is turned on, a table is added on the right hand side of the simulation area. In this table, the current distribution of states is shown and updated at every time step. Figure 20 shows an example simulation with activated statistics function. The proportion of cells in each state can be shown as absolute counts, or as percentages (relative).

In the statistics window, each state that occurs at least once is shown in one line. First the name of the state (text), then the color, and finally the number of cells in the lattice with this state.

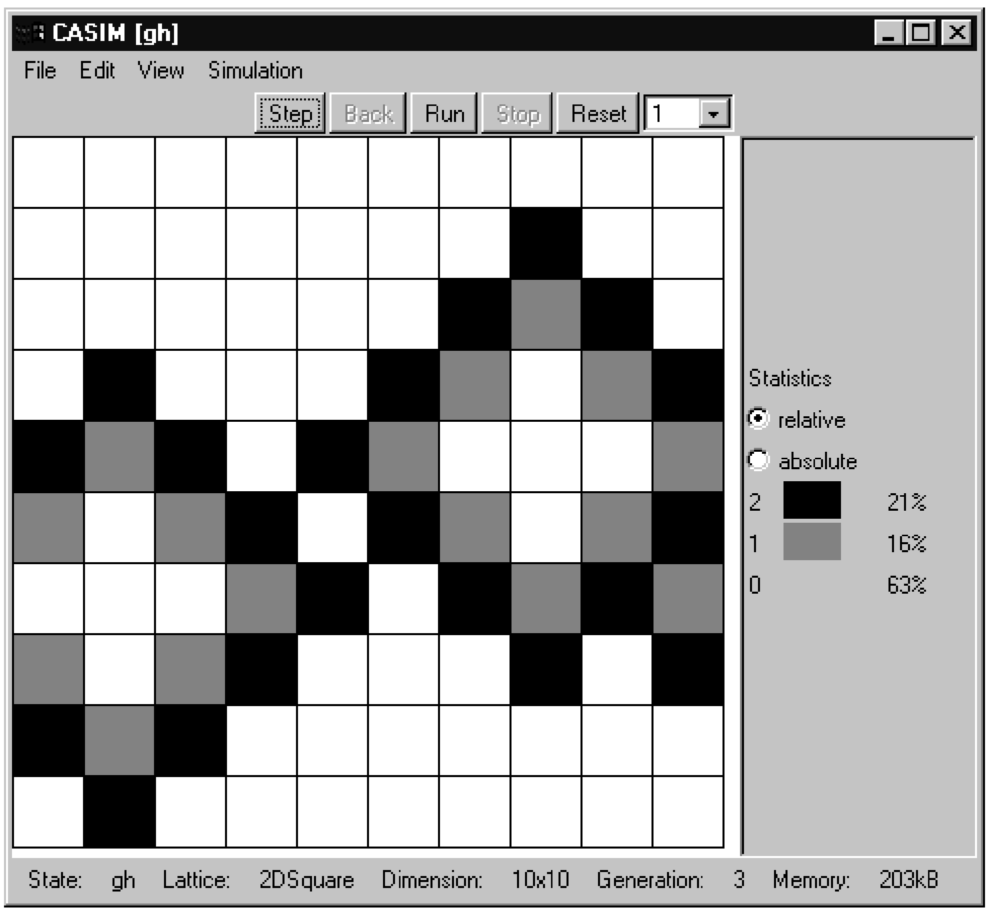


Figure 20: Statistics in CASim

**Clip Corner (3D)** Three-dimensional cellular automata are difficult to visualize. If all cells have an assigned color value, only the outside of the cube of cells is visible. To be able to have a look inside the simulation are, the option *View - Clip Corner* removes one octant of the simulation area from the display.

Another option is to assign a "dont' draw" color value to some states (return `null` in Java). These cells are then transparent and let the cells behind be seen. Unfortunately, this possibility comes with a severe performance penalty, since in this case all cells must be draw, while otherwise only the outside cells need to be drawn.

**History (1D)** For one-dimensioal automata, it is sometimes useful to show a time-space diagram with *View - History*, which shows older states below the current line of states.

#### 4.2.4 The menu Simulation

The simulation of a cellular automaton in CASimFrame can be controlled through the menu Simulation or through the buttons at the top. Both allow simulating the ca for a predefined number of steps before redrawing the new state. The simulation can also be run automatically. Reversible cellular automata can be run backwards.

**Step** The menu item *Simulation - Step* or the corresponding button instruct the system to simulate the CA for a given number of time steps and then redisplay the state. The button and menu item are only enabled if a CA has been loaded. A Keyboard-shortcut for this function is *CTRL + Right*.

**Back** Those cellular automata that are Block-CA (have class derived from `BlockState`), can possibly be reversible. If they are, *Simulation - Back* simulates one step backwards.

**Run** *Simulation - Run* starts a loop that simulates the CA for the number of time steps set in the control field on the top-right, then displays the state and possibly the statistics, and then continues the loop by simulating again.

**Stop** *Simulation - Stop* stops the loop started by *Simulation - Run*. The simulation stops after the current generation has been updated and does not continue until the next time a display is performed.

**Reset** To start again with the initial conditions, use the menu item *Simulation - Reset* or the corresponding button. This resets the lattice. If the initial conditions involve probabilistic choices, new choices are made. Therefore it often makes sense to repeatedly reset the lattice and observe the evolution.

**Steps** The number of steps to be simulated before the CA is displayed again can be selected with *Simulation - Steps* or the corresponding selector in the button panel. The current setting is displayed in the menu item, and is shown in boldface in the menu that pops up when you move to the right of the menu entry to select a new value.

## 4.3 The button controls

The different steps of the simulation can be controlled from the button panel at the top of the simulation window (see Figure 21). The buttons *Step, Back, Run, Stop, Reset* correspond to the menu items in the *Simulation* menu. The choice at the right allows you to select the number of time steps that should be simulated before the state is displayed again. Note that the simulation speed is mostly controlled by the time needed for drawing the CA, and not by the simulation of the CA.


Figure 21: Button controls of CASIM

The choice lets you select 1, 2, 3, 5, 10, 20, 50 or 100 as the number of steps simulated between successive displays. This choice is used for both the *Step* and the *Run* operations.

## 4.4 Status line

The status line at the bottom of the window displays the most important information items about the current cellular automaton. The items displayed are:

- *State*, the name of the CA or the Java class.
- *Lattice*, the dimension and geometry of the lattice.
- *Dimension*, the size of the lattice.
- *Generation*, the current time step of the CA.
- *Memory*, the amount of memory used by the simulation system.


Figure 22: The status line of CASIM

# 5 The JAVA Applet CASimApplet

An advantage of the programming language JAVA is that in addition to standalone programs, applications can be developed as applets which are executed by a web-browser without the need to install any software on the user's computer. A Java applet is part of a page described in the language HTML, which can conatin text, applets, images and other content.

Since Java applets are executed within a web browser, for security reasons many operations are forbidden. An applet is in general not allowed to read or write files on the user's computer. The applet may only load files from the server from which it was originally loaded. For this reason, the applet CASimApplet does not have all the possibilities of the full simulation system CASimFrame. CASimApplet provides the possibility to interactively define a cellular automaton or use a Java class that has been compiled on the server. The applet cannot use or compile a Java class from the user's computer. It is possible to change the lattice settings interactively, but the typical use of the applet is to demonstrate a cellular automaton using parameters specified in the HTML page, as shown in the HTML example code in Figure 23, which will appear similar to Figure 24 in a web browser.

The parameters of the applet, which describe the CA class to use, the lattice, and the appearance, i.e., all the parameters controlled by the menu items in CASimFrame, are passed as HTML parameters with the tag `param`:

```
<param name="Name of the parameter" value="value of the
                         parameter">
```

The different parameters are described in Table 1.

```
<applet code="de.tubs.cs.sc.cavis.CASimApplet"
archive="casimapplet.jar" height="400" width="640">
<param name="class" value="GH">
<param name="lattice" value="2DSquare">
<param name="x" value="21">
<param name="y" value="21">
<param name="neighborhood" value="vonNeumann">
<param name="radius" value="1">
<param name="initoption" value="0">
<param name="drawgrid" value="true">
<param name="drawtext" value="false">
<param name="clipcorner" value="false">
<param name="history1d" value="true">
<param name="showborder" value="0">
<param name="steps" value="1">
</applet>
```

Figure 23: HTML sourcecode with CASimApplet

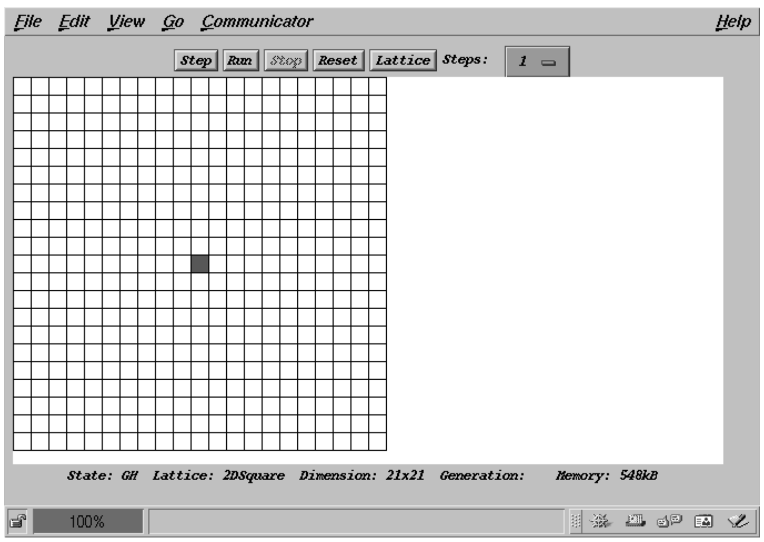


Figure 24: CASimApplet inside a web-browser

| Parameter | Description | Default value |
|---|---|---|
| class | name of the java class describing the CA, without the ending .class | – |
| stateset | the URL of an interactively defined CA. | – |
| lattice | dimension and geometry of the lattice, one of the values "1D", "2DTriangle", "2DSquare", "2DHexagonal" or "D" | 2DSquare |
| x | the width of the lattice | 10 |
| y | the height of the lattice | 10 |
| z | the depth of the lattice | 1 |
| neighborhood | the neighborhood, " Moore" or " vonNeumann" | Moore |
| radius | the radius of the neighborhood | 1 |
| initoption | integer number of the desired initial conditions | 0 |
| boundaryl | treatment of the left boundary, one of "Periodic", "Reflective", "Constant" | Periodic |
| boundaryr | treatment of the right boundary | Periodic |
| boundaryt | treatment of the top boundary | Periodic |
| boundaryb | treatment of the bottom boundary | Periodic |
| boundaryf | treatment of the front boundary | Periodic |
| boundarya | treatment of the rear (aft) boundary | Periodic |
| drawgrid | Should the grid be drawn? " true" or " false" | true |
| drawtext | Should the text for a cell be drawn? " true" or " false" | false |
| clipcorner | Should the left front corner be clipped? " true" or " false" | false |
| history1d | Should a space-time diagram be drawn? " true" or " false" | false |
| showborder | Number of border cells that should be drawn. | 0 |
| steps | Number of simulation steps per display step | 1 |

Table 1: The parameters of CASimApplet

# 6 The non-graphical program CASim

CASim is an application that can be started on the command line and does not require any graphical input or output. It can be used for timing measurements or to produce data for statistics or other automated analyses.

Since no graphical input is used, the complete specification of the CA must be done on the command line using parameters:

```
java de.tubs.cs.sc.casim.CASim options [-o file]
```

The options are listed in Table 2. An example call of CASim is

```
java de.tubs.cs.sc.casim.CASim  -s GH -l 2DS -x 10 -y 10 -g 20
```

or shorter

```
java casim -s GH -l 2DS -x 10 -y 10 -g 20
```

This will simulate 20 time steps of the cellular automaton which is described in the class GH. It will be simulated on a two-dimensional square lattice of size 10*10 with a vonNeumann neighborhood of radius 1.

| **Option** | **Description** | **Default** |
|---|---|---|
| -s | the name of the Java class | |
| -l | lattice geometry, 1D, 2DT, 2DS, 2DH, 3D | 2DS |
| -x | width of the lattice | |
| -y | height of the lattice | |
| -z | depth of the lattice | |
| -n | the neighborhood " Moore " or " Neumann " | Neumann |
| -r | dradius of the neighborhood | 1 |
| -g | number of time steps to be simulated | |
| -c | (on/off) neighborhood cache | on |
| -m | (on/off) measurement of speed and memory consumption | off |
| -o file | name of the file to be used for saving the final state | stdout |
| -h | print a short help text | |
| -b1p | boundary condition in 1st dimension (x) and positive direction (right) | Periodic |
| -b1n | boundary condition in 1st dimension (x) and negative direction (left) | Periodic |
| -b2p | boundary condition in 2nd dimension (y) and positive direction (top) | Periodic |
| -b2n | boundary condition in 2nd dimension (y) and negative direction (bottom) | Periodic |
| -b3p | boundary condition in 3rd dimension (z) and positive direction (rear) | Periodic |
| -b3n | boundary condition in 3rd dimension (z) and negative direction (front) | Periodic |

Table 2: Parameters of CASim

# 7 Cellular Automata using TableState

In addition to the possibility to write a Java class derived from `State` to describe a CA, you can also write a java class derived from `TableState`. This class is used for the interactively specifief automata, and you can also use it when programming in Java, since it already implements a number of the methods needed for a CA. The state in `TableState` is stored in an integer `condition`, which is defined in `TableState`.

The steps (1), (2) and (5) described in section 3 can be ommited, since the corresponding methods are already implemented in `TableState`. The methods `initialize(Lattice l)` (Step 3), `getColor()` (Step 4) and `transition(Cell c)` (Step 6) must be implemented. The methods `getColor` and `transition` need to be implemented as described in section 3. The method `initialize` already has an implementation in verb!TableState! which uses a table of probabilities to randomly initialize the lattice. To use this, you only need to implement the methods

```
public int getNrOfStates();
public double getProbability(int iState);
```

## 7.1 Number of states, Step(9)

In `getNrOfStates` the total number of states (counted from 0) muist be returned.

## 7.2 Probability of a state, Step(10)

`TableState` then calls `getProbability` for each state number between 0 and `getNrOfStates()` and initialized the lattice with the returned probabilities for the different states. Probabilities should be between 0 and 1, and should add up to 1.

If the probabilistic initialization is not desired, the method `initialize(...)` can be implemented. The lattice can also be initialized interactively using the popup-menus.

# 8 Block Cellular Automata using BlockState

Block cellular automata are those automata, where the rules specify how a block of cells should change their states together in one time step. The composition of the block (the block boundaries) can then be changed for different time steps to provide communication across block boundaries.

To simulate a block cellular automaton, derive a class from `BlockState` instead of `State`. Except for steps (5) and (6) the steps are the same as those for a regular CA. The methods `copy()` and `transition(Cell c)` are already defined as final in `BlockState` and cannot be changed, because they are not needed for block-CA. In addition to steps (1) to (4) you need to implement the methods

```
int [][][] getBlocks(int dim)
void transition(States[] states)
void reversetransition(States[] states)
```

of `BlockState` in steps (11) to (13).

## 8.1 Defining a block pattern, Step(11)

First you need to define a pattern of blocks. The simulation system calls the method `int [][][] getBlocks(int dim)` once with the dimension of the current lattice. You should return an array of indices consisting of a list of blocks, which are then used cyclically in subsequent time steps. Ech block consists of an array of addresses, each of which is a vector of `dim` integers. All blocks should have the same number of elements, and they should be rectangular in shape.

The Margolus blocks, which consist of blocks of four cellswhich are shifted across the diagonal for subsequent steps, are defined as shown in Figure 25.

```
static int [][][] block2D = {
    // first block
    { {0,0}, {0,1}, {1,0}, {1,1} },
    // second block
    { {1,1}, {1,2}, {2,1}, {2,2} }
} ;
```

Figure 25: Definition of Margolus blocks in JAVA

In the method `getBlocks()` you can then return `block2D` directly. CASim determines the number and size of the blocks and tiles the lattice with these blocks. The cells belonging to one block are then stored in one array and given as arguments to the functions `transition` and `reversetransition`.

## 8.2 State transition function for block-ca, Step(12)

The transition of one block from one time step to the next is coded in the method `transition(BlockStates[] states)`. This method must be implemented in your Java class, and must calculate the new state for all cells in array `states`, which contains the cells of one block. You are allowed to access all fields of every element of the block, but not access any neighbors with `getNeighbors()` or similar functions. Of course, the elements of the array must be type cast to your subclass before you can access any fields.

### 8.3 Reverse transition function, Step(13)

If the transition function is reversible, the whole CA is reversible. In this case, you can implement the method `reversetransition` to be able to reverse the simulation.

# 9 Asynchronous Cellular Automata

A generalization of cellular automata is to use an asynchronous updating method. In these methods, only one cell is updated at a time, and the new state becomes immediately visible to the neighbors. Different asynchronous methods differ by the order in which the cells are selected for updating. An overview over different asynchronous methods and an analytical investigation into their statsitical properties can be found in Schönfisch (1999).

Here we simply demonstrate a number of options. The simplest option (1) is to sweep through the lattice of cells line-by-line. This order also appears in the Gauss-Seidel updating method in numerical mathematics. In cellular automata this order usually leads to large artifacts, since it violates the symmetry of the CA: Information can flow from left to right over arbitrary distances in a single time step (it is sweeped along), whereas information flow from right to left proceeds only by one neighborhood radius in one time step (as in synchronous cellular automata).

The second possibility (2) is to use a fixed order of updating of all cells within each time step. To ensure that each cells gets updated exactly onece in each time step, this order should be a random permutation. Random permutations can be generated by the standard shuffling algorithm described by Knuth:

```
for (i=0; i<N; i++)
    a(i] = i;
for (j=N-1; j>0; j--) {
    i = choose(0,j);
    exchange a[i] and a[j].
}
```

where `choose(0,j)` uses any good random number generator to select a number between 0 and j.

Note that the line-by-line sweep is just one example of a random updating oder. It occurs with probability $\frac{1}{N!}$ in a lattice of $N$ cells. This is a very small probability, but some of the features of this special case appear (at least in local regions) in almost all random permutations: in some regions, information propagates much faster in certain directions than in others. Since the updating order remains unchanged between time steps, this becomes very noticable after a number of steps.

The third possibility (3) is to use a different random permutation for each time step. The fourth possibility (4) is to relax the requirement that each cell should be updated exactly one in each time step, and just randomly select the next cell to be updated. Then $N$ such updates of randomly selected cells can be considered as one time step. In this model, it is possible that the same cell is updated twice in succession before any of its neighbors are updated.

A fifth possibility (5) is to assign a waiting time to each cell and update the cells in the order in which the waiting times expire. When a cell is updated, a new waiting time is generated. The waiting times could be generated from any distribution, but to avoid rollbacks, times should be positive. A suitable distribution is the exponential distribution. To determine the next cell to be updated, a list of cells and waiting times should be kept in a heap, which is a datastructure which allows fast access to the element with minimum value, and O($logN$) insertion of new elements.

In JCASim, asynchronous cellular automata can be used by specifying the following class to be used as the `Lattice`, which is currently oionly implemented for 2-D square lattices: `de.tubs.cs.sc.casim.Lattice2DAsync` This is specified in the applet by the parameter `<param name="latticeclass" value="de.tubs.cs.sc.casim.Lattice2DAsync">` and in the application by specifying the following command line option: `java casim -l de.tubs.cs.sc.casim.Lattice2DAsync`

The different possible asynchronous updating methods are then selected by setting different lattice options (1..5) in the lattice dialog.

# 10 Feedback

The simulation system CASim has been written to simplify using cellular automata. It also aims to show the many possibilities of simulation with cellular automata. If you have suggestions for improvement, have developed interesting application, or just want to comment on the program, please contact me using E-mail at `J.Weimar@tu-bs.de`. I hope the program will be useful for you.