Agent Based Modeling
-
Different Scenarios On The Sugarscape

A contribution related to the course

Open Source Software

WWU Münster - MSc. Geoinformatics

SS 2013

Version 1.0

by Friedrich Müller

7.09.2013

Contents

1. Introduction

This work deals with the implementation of an Agent Based Model (ABM) concept with the TerraME modeling language which is an extension of Lua. Lua is an open source, interpreted language with extensible semantics. TerraME uses the LUA extensibility mechanisms to include new data types and functions for the development of spatial dynamical modeling.
For further information about Lua and TerraMe see [4, 5, 6]
The scenarios that are implemented in this tutorial are based on the examples given in the book of Epstein and Axtell. ABM is a method that is based on individuals that are represented by the agents. The method is used for model building and creating simulations. An important aspect of ABM is that the concept allows to discover the connection between micro and macro level .
So, it helps to observe the behavior of the system level that results of the behavior of single individual behavior. And in which way the behavior of the agents influence the system environment. Or saying in other words to discover complex rules that are created from simple behavior rules. So you can say that the the whole system is more than the sum of the parts of the individual behavior.
An interesting aspect is to see ABM as an ' CompuTerrarium' that allows the developer to notice the results of changing setting rules like governing trade or inheritance.
ABM application areas are traffic simulations, social networks, economic systems or segregation models.

1.1 The Sugarscape

As the book title of Epstein and Axtell reveals to us the issue we deal with is to build up artificial societies from bottom up. The sugarscape is in this connection a landscape with areas that contain different amount of sugar. There are areas with no sugar as well as high peak sugar areas. The inhabitants, the agents of the sugarscape, are individuals that want to eat sugar. They can collect a certain amount of sugar, burn sugar according to their initiation metabolic rate. Furthermore the sugar amount is essential for the agent. If an agent has no sugar it dies. In order to get sugar the agent has to move. This takes place according to a movement rule. The agent's movement is restricted by its vision that allows it to scan the neighbor field within a predefined area.
In summary there are three basics in the sugarscape model: Agents, environment, rules. Each agent has initiation states like wealth, age and behavior rules e.g. movement. These states and rules are not static and can change over the lifetime of an agent e. g. through interaction with other agents, its own behavior, or according to the space the agent moves. As in real life the agent individual is born, able to have offspring and die. The environment has to be seen separated from the agents. An environment can be an area with places of different sugar appearance and where the resource sugar disappears or is renewed under certain rules. Agents interact on the environment with movement or with the environment e.g. collecting sugar. An environment could also be a communication network that geometry could change over time. Rules can be set differently and determine the behavior of agents and sites of environment. It can be distinguished between *agent-environment* rules e.g. in which way an agent collects the sugar of a place, *environment-environment rules* e.g. the sugar regrowth function of the sites of the environment and *agent-agent rules* like for mating,combat, trade or cultural transmission.

In the following we want to create a new society from bottom up. In this work we begin with the creation of the sugarscape environment and then populate it with agents. We define besides basic rules e.g. movement, environment-environment rules like seasonal sugar growback rules and agent-agent rules like mating.
The models are oriented on the described examples from Epstein and Axtell and sometimes a bit modified like e.g. that the agent take instead of the nearest high sugar value cell a random high sugar value . The main scenarios of the book that are considered in this work are movement, resource gathering, migration, mating, inheritance and cultural transmission. The given code excerpts should be seen as examples. Moreover the given examples are not the complete code of a scenario, the focus is only on the key code parts. For getting the executable examples please see the example code files.
Furthermore, for getting a more detailed view on types and functions of TerraMe you can look at the TerraME documentation [2]. Before using this tutorial, the reader should first install TerraME. The software and Development kits, instructions for installation and running within different development environments can be found on www.terrame.org.

2. Building a society model from bottom up

At the beginning we need to create the sugarscape in a spatial way. We have to design a 'world' in which our scenarios can take place. So we have to generate a spatial distribution of sugar.
Therefore we create an object of type *CellularSpace.* With this object we can read data from an existing database in which the spatial entities with different attributes like predefined number of sugar or maximum sugar value a cell can reach is stored. We need to define where the database can be found, the name of the theme within the database and which attributes should be read (Code 1). The cs CellularSpace connects to the Microsoft Access database and reads out for all cells the attributes maxSugar and sugar. The last line creates a von Neumann neighborhood for each cell. This means that each cell has exactly one neighbor in north, south, east and west direction.

```
cs = CellularSpace {
        database = "c:\\sugarscape.mdb",
        theme = "sugarscape",
        select = {"maxSugar", "sugar"}
}
cs:createNeighborhood{strategy = "vonneumann"}
```

Code 1: Defining a CellularSpace database connection

Another option, if we have no existing database, is to create the sugarscape from sketch (Code 2). Here we create with xdim and ydim a rectangular *CellularSpace.*
This example equates a square (50x50) of 2500 cells.

```
cs = CellularSpace {
        xdim = 50,
        ydim = 50}
```

Code 2: Defining a CellularSpace

Now we have the CellularSpace but the attributes of each cell are still missing.
First we have to define and set up the the cell attributes sugar and maxsugar for each cell. The second order function forEachCell set up for each cell of the CellularSpace (cs) the mentioned attributes.
As a next step we have to define regions with different maximum sugar value on the sugarscape. With the function getCell() of the CellularSpace we declare the maxSugar value of the cell with the named coordinates as 4. The coordinates are chosen that one sugar high peaks is in the northeast area and one in the southwest area.
Then with the help of loops we create terraces of equal maxSugar value (Figure 1).

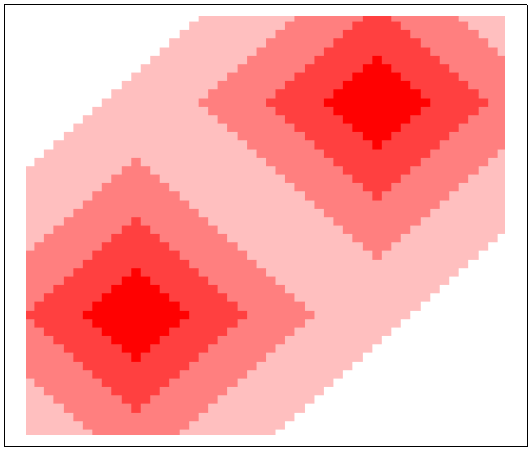


Figure 1: Terrace-formed sugar areas with high value regions in northeast and southwest

The function synchronize() synchronizes the Cellularspace and updates the past values with the current attributes. Therefore we check for each cell of the Cellularspace with the help of forEachCell() function the neighbors of the von Neumann neighborhood with the forEachNeighbor() function . If the maxSugar value of the previous time step is equal the condition value in the first loop or in the following loops is bigger or equal the value then the neighbor cells attribute has the same value (Code 3).

```
forEachCell(cs, function(cell)
        cell.maxSugar = 0
        cell.sugar = 0
end)



cs:getCell(Coord{x = 11, y = 35}).maxSugar = 4
cs:getCell(Coord{x = 36, y = 10}).maxSugar = 4

for i = 1, 5 do
        cs:synchronize()
        forEachCell(cs, function(cell)
                forEachNeighbor(cell, function(cell, neighbor)
```

```
                        if neighbor.past.maxSugar == 4 then
                                cell.maxSugar = 4
                        end
                end)
        end)
end


for i = 1, 6 do
        cs:synchronize()
        forEachCell(cs, function(cell)
                forEachNeighbor(cell, function(cell, neighbor)
                        if neighbor.past.maxSugar >= 3 and cell.maxSugar == 0 then
                                cell.maxSugar = 3
                        end
                end)
        end)
end

for i = 1, 7 do
        cs:synchronize()
        forEachCell(cs, function(cell)
                forEachNeighbor(cell, function(cell, neighbor)
                        if neighbor.past.maxSugar >= 2 and cell.maxSugar == 0 then
                                cell.maxSugar = 2
                        end
                end)
        end)
end

for i = 1, 10 do
        cs:synchronize()
        forEachCell(cs, function(cell)
                forEachNeighbor(cell, function(cell, neighbor)
                        if neighbor.past.maxSugar >= 1 and cell.maxSugar == 0 then
                                cell.maxSugar = 1
                        end
                end)
        end)
end
```

Code 3: Generating sugar value terraces

After having created an initial distribution of sugar the next task is to bring life on the sugarscape.
We have to define inhabitants that live on the predefined sugarscape that is a cellular space. As a first model the agents have the ability to gather sugar and consume it. For this we use the type Agent. An Agent is an autonomous individual that is described by its attributes and behavior. The attribute of the sugarAgent is age and further attributes

wealth, metabolism and the maximum age an agent can reach, are defined randomly with a certain value the function init() is used
to initialize an Agent when it enters in a given Society

The functions init() and execute() have a single parameter representing the sugarAgent itself. The function execute() describes the actions executed by the prey at each time step.
In this example (Code 4) the execute () function does following:
It is checked in which cell the agent is located and its neighbor cells are examined according to the von Neumann neighborhood the cell where the cell with the highest sugar value is located. Then the agent moves to this cell. If there are neighbor cells of equal sugar value then the agent moves to a random neighbor cell.

-First the Agent cell is defined and is stored in bestOptions array

The agent cell that is destinated by the function getCell() .

-Then the forEachNeighbor() function checks which of the agent neighbor cells has the

highest sugar value. If one is found then it defines itself as best option.

-If there are cells of equal highest sugar value then the destination cell is chosen randomly

Furthermore, each time step the agent gets one unit older and its wealth is calculated new. The agent's new wealth is its past wealth reduced by its metabolism and increased by the earned sugar of the new cell. The agent earns all the sugar and the sugar value of the agent cell becomes zero.
In the case that the agent's age is over the predefined maximum age or its wealth is smaller equal 0 then a new offspring agent with the same attributes and behavior like the agent is reproduced by using the function reproduce(). This new agent is placed in the Cellular space randomly with the function enter() of the agent. As an argument a sample cell of the cellular space is used. The 'parent' agent dies by using the function die().

```
sugarAgent = Agent{
	age = 0,
	init = function(self)
		self.wealth = math.random(5, 25)
		self.metabolism = math.random(1, 4)
		self.maxAge = math.random(60, 100)
	end,
	execute = function(self)
		bestOptions = {self:getCell()}
		forEachNeighbor(self:getCell(), function(cell, neighbor)
			if neighbor.sugar > bestOptions[1].sugar then
				bestOptions = {neighbor}
			elseif neighbor.sugar == bestOptions[1].sugar then
				bestOptions[table.getn(bestOptions)+1] = neighbor
			end
		end)
```

```
            bestOption = bestOptions[math.random(table.getn(bestOptions))]
            self:move(bestOption)

            self.age = self.age + 1
            self.wealth = self.wealth - self.metabolism + self:getCell().sugar
            self:getCell().sugar = 0

            if self.age > self.maxAge or self.wealth <= 0 then
                son = self:reproduce()
                son:enter(cs:sample())
                self:die()
            end
        end
}
```

Code 4: Movement rule

After having the type Agent generated it is necessary to define where at the start of the model it is located and what is the amount of the agent population. For this we create a society. The type agent that we have defined before can be seen as a pattern or a kind of blueprint that defines the members of the society. The amount of society members is in this example 50.
Furthermore, an environment is composed by the cellular space and the society. Using the function *createPlacement() agents are put in the cellular space* using a random strategy with maximum 1 agent in each cell. (Code 5)

```
soc = Society{
        instance = sugarAgent,
        quantity = 50
}


e = Environment{cs, soc}

e:createPlacement{
strategy = "random",

max = 1}
```

Code 5: Creating Society and environment with random initial placement of agents

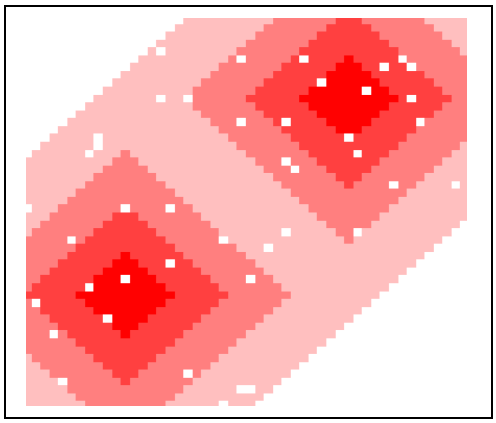
In addition we can use a sugar growback rule (Code 6) that increase the current sugar value per one unit of each cell of the cellular space if the current sugar value is smaller than the cell's maximum sugar value. This is an example for an environment-

environment rule.
The function growSugar () can be called in the timer type for executing it each time step (see example file Sugarscape_Startexample_Without-db.lua).

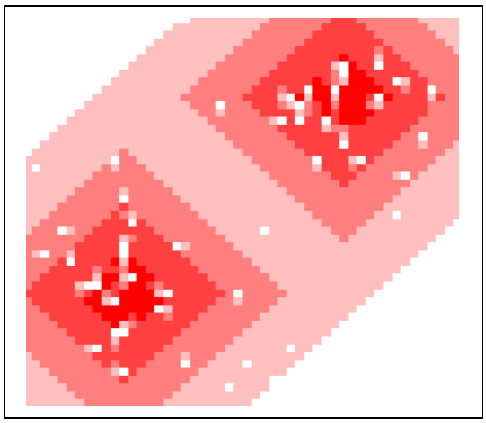
```
growSugar = function()
        forEachCell(cs, function(cell)
                if cell.sugar < cell.maxSugar then
                        cell.sugar = cell.sugar + 1
                end
        end)
end
```

Code 6: Sugar growback rule

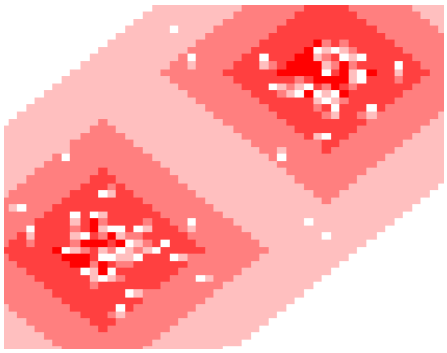
The initial distribution of agents and the situation of a simulation at different time steps are shown in Figure 2 . The spatial distribution of sugar is shown in different red tone colors. A more intense red color stands for a higher sugar value of the cell. The maximum sugar value is between 0 and 4. At the beginning 50 agents enter the cellular space. The edge areas have zero sugar value and are displayed white. The location of the agents is displayed by the white cells within the red area because as an agent moves to a cell it collects all the sugar. As the simulation proceeds we can clearly see that the agents move towards the areas of high sugar value.



a) Initial distribution of agents



b) Distribution of agents after 20 time steps.



c) Distribution of agents after 60 time steps.

Figure 2: Agent movement at different simulation time

As a next step we care about the movement rule of an agent. In the former example the agent was just allowed to move within its von Neumann strategy defined neighborhood. Now we want to invent the idea of vision to the agents (Code 7).
Here we just consider the cells within the vision area. This is made through defining the area by coordinates. First the code search or the best solution in east-west direction that is saved in the variable bestOption1. Then we do the same for the north-south direction that is saved in bestOption2. If the best solutions of north-south and east-west has equal sugar value then the destination cell is selected randomly in bestOption3. All in all the agent moves to one of the highest sugar value cells in the north, south, east and west direction within vision range.

Vision means that an agent can see a predefined number of cells in the four principal lattice directions north, south, east and west. So agents have no diagonal vision.

```
        execute = function(self)
        bestOptions = {self:getCell()}
        bestOptions2 = {self:getCell()}

-- if one neighbor has more sugar then bestOptions is the neigbor cell.


--vision
        vision=10
-- only the cells in east, west direction are considered within the vision area
        forEachCell(cs, function(cell)
--exclude the current position cell
        if cell.x ~= self:getCell().x then
                if cell.x< self:getCell().x+vision and cell.x> self:getCell().x-vision then

                        if cell.y==self:getCell().y then
```

```
-- defines the neighbor cell with the highest sugar value in bestOptions
                        if cell.sugar > bestOptions[1].sugar then

                                bestOptions = {cell}

-- if neighbor has equal sugar then the neigbor is saved in the array and get a own index
number
                        elseif cell.sugar == bestOptions[1].sugar then

                                bestOptions[table.getn(bestOptions)+1] = cell --getn,
which returns the size of an array

                                end
                        end
                end
                end
        end)
-- this choose a neighbor cell  randomly.
        bestOption1 = bestOptions[math.random(table.getn(bestOptions))]



-- N,S vision cell area

        forEachCell(cs, function(cell)
--exclude the current position cell
                if cell.y ~= self:getCell().y then
                  if cell.y< self:getCell().y+vision and cell.y> self:getCell().y-vision then

                        if cell.x==self:getCell().x then

-- defines the neighbor cell with the highest sugar value in bestOptions

                                if cell.sugar > bestOptions2[1].sugar  then

                                        bestOptions2 = {cell}


-- if neighbor has equal sugar then the neigbor is saved in the array and get an own index
number
                                elseif cell.sugar == bestOptions2[1].sugar then
                                        bestOptions2[table.getn(bestOptions2)+1] = cell
--getn, which returns the size of an array

                                end



                        end
                end
```

```
                end
        end)

-- this choose a neighbor cell  randomly.
        bestOption2 = bestOptions2[math.random(table.getn(bestOptions2))]
        if season_time>1 then
                if bestOption1.sugar>bestOption2.sugar then
                        self:move(bestOption1)
-- the agent moves to the neigbor with the highest sugar, if there are more equal high
sugar value cells the movement is randomly.
                else
                        self:move(bestOption2)
                end

                if bestOption1.sugar==bestOption2.sugar then


                        bestOptionChoice = {bestOption1, bestOption2}
                        bestOption3 =
bestOptionChoice[math.random(table.getn(bestOptionChoice))]
                        self:move(bestOption3)
                end
        end
```

Code 7: Movement rule with vision

Now, that we can create agents with different vision, we can think about using two types of agent. A bird-like one that has a high vision e.g. 5 and low metabolism like a value between 1 and 2 and a bear like one that has a low vision e.g. 2 and a high metabolism like a value between 3 and 4. Therefore we first have to generate the two types of agents and create a society of each type with a quantity here 25.
After that we can add them to the environment and place them, as in this case ,randomly (Code 8).

```
bears  = Agent{
        age = 0,
        vision= 2,
        init = function(self)
                self.wealth = math.random(5, 25)
                self.metabolism = math.random(3, 4)
                self.maxAge = math.random(60, 100)
        end,
        execute = function(self)
         movement()
        end
}
birds  = Agent{
        age = 0,
        vision= 5,
```

```
        init = function(self)
                self.wealth = math.random(5, 25)
                self.metabolism = math.random(1, 2)
                self.maxAge = math.random(60, 100)
        end,
        execute = function(self)
         movement()
        end
}

n_birds=25
        soc_birds = Society{
                instance = birds,
                quantity =n_birds
        }

n_bears=25
        soc_bears = Society{
                instance = bears,
                quantity =n_bears
        }

e = Environment{cs, soc_birds,soc_bears}

e:createPlacement{strategy = "random"}
```

Code 8: Creating bear and bird like agents, adding the agent to society and environment

The next topic we deal with is migration. In this example the initial distribution of agents is in a block in the southeast corner of the cellular space. The placement is according Code 9. Here the placement strategy void is used. This strategy allows the developer to define the initial placement of the agents by a function. In the example code the agent placement starts at x=0 and y= 49. The if condition avoids that agents initial x coordinate is bigger than 10. So an amount of 50 agents would create an agent rectangle of 10 agents width and 5 agents height. See also Figure 3. Notable is the diagonal movement of the agents during the simulation.

```
x1=0
y1=49
e = Environment{cs, soc}

e:createPlacement{strategy = "void"}

forEachAgent(soc, function(agent)

        if x1==10 then
                x1=0
                y1=y1-1
        end
        c = Coord{x = x1, y = y1}
        f = {cs:getCell(c)}
        agent:enter(f[table.getn(f)])
        agent:getCell()
        x1=x1+1
end)
```

Code 9: Initial placement strategy void that allows to place the agents to a destination place

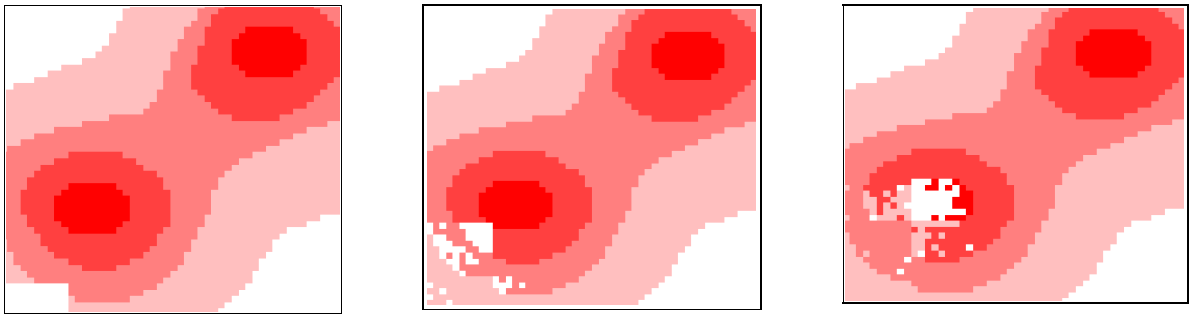


Figure 3: Agent movement at different simulation time with high metabolity rate 4 and vision =10

Another example for migration is the use of a seasonal sugar growback rule. Therefore we have to generate artificial seasons. For this the sugarscape is divided into a northern and a southern part. Our equator is defined by y= 25. In the first season it is summer in the north and winter in the south. A summer is defined that there the sugar grows back one unit each time step whereas in the winter the growback rate is only 1/8 sugar unit. The season changes every 50 years. In order to set the rule for all cells of the cellular space we use the for forEachCell() function.
The example code is shown in Code 10

```
growSugar = function()
        forEachCell(cs, function(cell)
                if season_time<50 then
                        if cell.y>= 25 then
                                if cell.sugar < cell.maxSugar then
                                        cell.sugar = cell.sugar + 1/8
                                 end
                         elseif cell.y< 25 then
                                if cell.sugar < cell.maxSugar then
                                        cell.sugar = cell.sugar + 1
                                end
                        end
           end
                if season_time>=50 then
                        if cell.y>= 25 then
                        if cell.sugar < cell.maxSugar then
                                        cell.sugar = cell.sugar + 1
                            end
                      elseif cell.y< 25 then
                                if cell.sugar < cell.maxSugar then
                                        cell.sugar = cell.sugar + 1/8
                                end
                      end
                end
                if season_time ==100 then
                        season_time=0
                end
            if cell.sugar > cell.maxSugar then
                        cell.sugar=cell.maxSugar
            end
        end)
end

season_time=0
t = Timer{

        Event{action = function()
        growSugar()
        season_time=season_time+1
        end},
        Event{action = soc},
        Event{action = updateCell},
        Event{action = cs},
        Event{action = cell}
}

t:execute(100)
```

Code 10: Seasonal sugar growback rule

In the example of Figure 4, 100 agents with vision 20 and metabolism rate 3 start at the southeast corner of the Cellularspace. As initial season it is summer in the north (sugar regrow of 1) and winter in the south (sugar regrow of 1/8). The season change after 50 time steps.
First the agents move to the sugar hot spot in the southeast and collect all the sugar then they move to the northern hot spot and they stay there until the summer period of the north region.
When the season change and it is winter in the north and all the sugar resources are exhausted then the agents begin to migrate to the south region where the sugar offer is higher during summer.

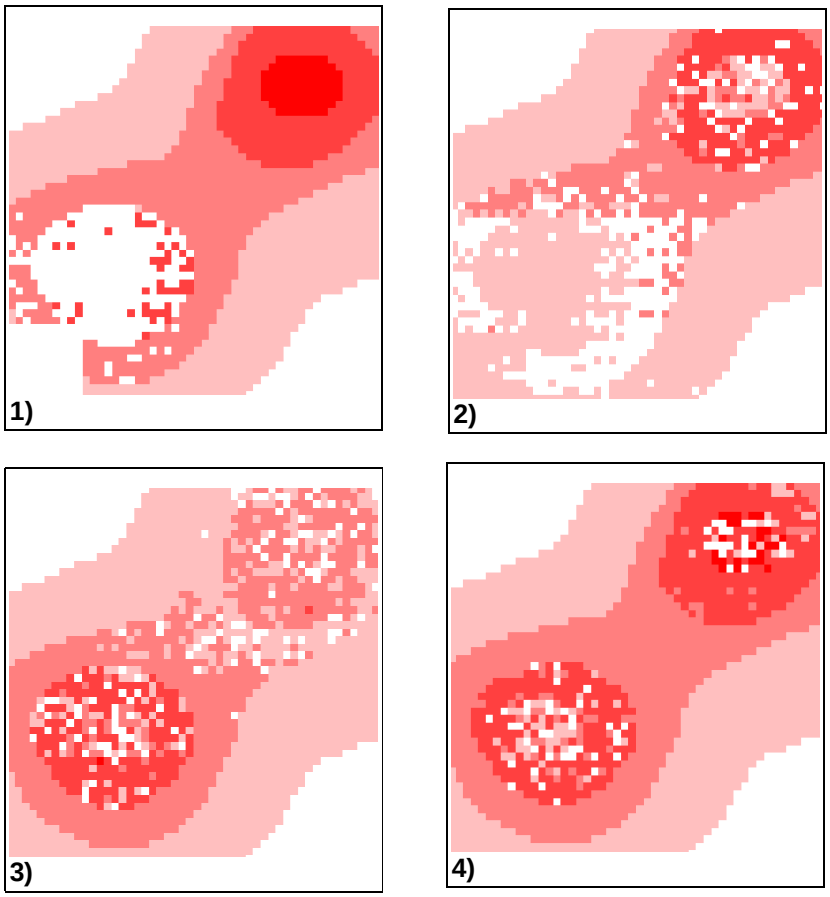


Figure 4: Agent movement influenced by different seasonal sugar regrow conditions

As a next step we want to create a counter-player to the attribute sugar that influences an agent in a negative way – a pollution value. The negative influence could be added in different ways like if the pollution value of an agent grows it could reduce its vision for example. In our example the pollution value influence the movement behavior of the agents on the sugarscape. Therefore a sugar pollution ratio is invented. The ratio is calculated as the cell sugar/ (1-cell pollution). So the agents don't look now for the cell with the highest sugar value within their vision but for the cell with the highest sugar- pollution ratio.
In the given example in Code 11 there is no pollution in the first 10 time steps. After that if an agent moves to a cell the pollution of that cell is increased by the value of the agent's pollution. Here the agent pollution is one eight of its metabolism
This procedure takes place until the time step 30 is reached then a diffusion of the cell pollution occur each cell in the cellularspace get the average pollution of its von Neumann neighbors. This value is kept constant in the following time steps.

```
-- define sugar/pollution ratio
forEachCell(cs,function(cell)
        cell.p_ratio=cell.sugar/(1+cell.pollution)
end)

sugarAgent = Agent{

        age = 0,
        init = function(self)
                self.wealth = math.random(5, 25)
                self.metabolism = math.random(1, 2)
                self.maxAge = math.random(60, 100)
                self.pollution=0
        end,
        execute = function(self)
           …
           self:move(cell)
           self.pollution=self.metabolism/8
           if season_time>=10 and season_time<30 then
                self:getCell().pollution= self:getCell().pollution+ self.pollution
                self:getCell().p_ratio= self:getCell().sugar/(1+self:getCell().pollution)
                end
           …
        end
}

updateCell = function()

 if season_time==30 then
  forEachCell(cs, function(cell)
    sumPollution=0

        forEachNeighbor(cell, function(cell,neighbor)
```

```
            sumPollution= sumPollution+neighbor.pollution
        end)


  averagePollution=sumPollution/4
  cell.pollution=averagePollution
  cell.p_ratio=cell.sugar/(1+cell.pollution)

 end)
end
end
```

Code 11: Example of a pollution rule

As so far, the agents influenced the behavior of each other indirectly. Agents move over the cells and what they find is the result of the other agents that already visited the place like the lack of sugar of a cell or a high pollution rate. Now we want to have a look at some more direct interaction between agents like mating or cultural transmission.
In our mating example we use agents types of men and females. The mating rule is that if a man cell meets a female agent cell that has to be a direct neighbor within the von Neumann neighborhood and both are within a reproduction age then a new agent is born. The child tries to enter first to an empty neighbor cell of the men and if there is no free space it tries to find an empty place in the female cell neighborhood. If there is no free space then no new agent is added to the cellular space. The gender of the offspring is defined randomly. The probability is 50% that a child is a man or female agent. The beginning of the reproduction is defined for both gender randomly between 12 and 15 years. The end of reproduction is reached by females at a random value between 40 and 50 years and by men at a random value between 50 and 60 years (Code 12).
A further variety of the mating rule would be e.g. that a reproduction is only possible if men or female agents have a certain value of wealth.

```
count=0
neighborchoice={}



forEachNeighbor(self:getCell(), function(cell, neigh)
        if neigh.name=="female" and self:getCell().name=="man" then
                if neigh.age>neigh.reproductionstart and
self:getCell().age>self:getCell().reproductionstart then
                        if neigh.age<neigh.reproductionend and
self:getCell().age<self:getCell().reproductionend then

                                reproductionProbability=math.random(0,1)
```

```
                        if reproductionProbability == 0 then
                                forEachAgent(neigh, function(agent)
                                if agent.name == "female" then
                                        son= agent:reproduce{}

                                end
                                 end)

                else

                        son =
self:reproduce{age=0,inheritance=INHERITANCE,dad=ID1,mum=ID2}

                        end

                        count=count+1

-- if all neigbors of the man cell are occupied look for a empty place at the female
neighbors' cell
                                if count==4 then
                                        forEachNeighbor(neigh, function(cell, neighbor)
                                        if neighbor.name=="EMPTY"  then


        neighborchoice[table.getn(neighborchoice)+1]=neigh


        son:enter(neighborchoice[math.random(table.getn(neighborchoice))])
                                                end
                                        end)
                                end

                                if neigh.name=="EMPTY" and count<4 then
                                neighborchoice[table.getn(neighborchoice)+1]=neigh

        son:enter(neighborchoice[math.random(table.getn(neighborchoice))])
                                end

                        end
                end
        end
end)
```

Code 12: Reproduction rule

Another concept that can be added to the sugarscape is the idea of inheritance (Code 13). In our case we implemented it in that way that each child has an inheritance value that is equal to the sum of the wealth value of each parent agent at the time of reproduction. After one parent dies each of its offspring earns the same part of the whole inheritance.

Therefore we first have to invent an agent id value that is stored during the reproduction process. Then if an agent dies we have to check the number of its children and following distribute the inheritance at equal parts to the offspring.

```
IDinheritance=0

if self.age > self.maxAge or self.wealth <= 0 then

	IDinheritance=self.id

	self:die()

	childcount=0
	forEachCell(cs, function(cell)
		forEachAgent(cell,function(agent)
			if agent.mum== IDinheritance then
				childcount=childcount+1
			end
			if agent.dad== IDinheritance then
				childcount=childcount+1
			end
		end)
	end)

	forEachCell(cs, function(cell)
		forEachAgent(cell,function(agent)
			if agent.mum ~=0 and agent.dad~=0 then
				if agent.mum== IDinheritance then
					print("Child"..childcount)
					print("inheritance"..agent.inheritance)
					print("W"..agent.wealth)
					agent.wealth= agent.wealth+agent.inheritance/childcount

					print("W"..agent.wealth)
				end
				if agent.dad== IDinheritance then
					agent.wealth= agent.wealth+agent.inheritance/childcount
				end

			end
		end)
	end)
end
```

Code 13: Inheritance rule

In the following section we add another behavior rule to the sugarscape. A cultural transmission rule. For describing the membership of a culture we use cultural tags. This cultural tag is a binary code string consisting of zeros and ones. Each agent type has an own cultural tag format. Agents can change the cultural tags of other agents. The cultural transmission rule from our example lets the cultural attribute of bird agent change. If the agent has a bear agent as a neighbor and the focused character of the cultural tag differs between the two agents, then the selected character is changed. If the tag is equal no changing occurs. For creating this behavior every time a bear agent meets a bird one the place of the cultural tag that should be compared is selected by a combination of the math. random() and string.len() function. The string function gives back the number of string length that is in our case 11 for bird and bear agents. So a character from 1-11 is selected randomly. For making a comparison possible the function string.sub() is used that gives out a character at a certain place of a string. Then, this character is compared. If it is equal no cultural transmission takes place, if it differs the character of the tag from the bird agent is changed into the character of the bear agent. This is made by using the function replaceChar () that needs three values as input a string, the place number of the string, and the new character (Code 14).
Furthermore the agents are visualized in different colors. The color depends on the number of zeros the cultural tag of an agent contains. For getting the number zeros in a string we use the function char_count(). Finally the number of zeros that are appearing in a string are with an if-condition categorized.
The color distribution in the model is:

Number of zeros within a tag	Color of the agent cell
0-3	blue
4-7	green
7-11	red

We can see that cultural transmission progress takes place during a simulation (Figure 5). Here the behavior of three bird agents and one bear agent is observed

```
function char_count(str, char)
    if not str then
        return 0
    end

    local count = 0
    local byte_char = string.byte(char)
    for i = 1, #str do
        if string.byte(str, i) == byte_char then
            count = count + 1
        end
    end
```

```
            return count
        end
---------- Source: http://amix.dk/blog/post/19462

-- function that replace a character with a predefined value within a string at a certain place
        function replaceChar(str,idx,rep)
            local pat
            if idx==1 then
                pat="^.(.*)$"
                return string.gsub(str,pat,rep.."%1")
            else
                pat="^(" .. (".":rep(idx-1) .. ").(.*)$"
                return string.gsub(str,pat,"%1"..rep.."%2")
            end
        end
---------- Source: http://forum.luahub.com/index.php?topic=2646.0




-- culture time is  the time a agent stays at his current culture
        culturetime=0

        forEachNeighbor(self:getCell(), function(cell, neigh)
                place= math.random(1,string.len(self.culture))
                string_bear=string.sub(self.culture, place,place)
                if neigh.name=="BIRD" then
                        forEachAgent(neigh,function(agent)
                                if agent.name=="bird" then
                                        test=agent.culture
                                        string_bird=string.sub(test, place,place)




                        if string_bird ~=string_bear and season_time>culturetime then
                                                print(string_bird)
                                                print(string_bear)
                                                x=tostring(string_bear)
                                                check=replaceChar(test,place,x)
                                                culturetime=season_time+1
                                                print("Change CULTURE")
                                                print(test,check)
                                                agent.culture=check
                                        end
```

```
                        colorcheck= char_count(check, 0)
                            if colorcheck<=3 then
                                agent.state=BLUE
                                agent:getCell().state=BLUE

                            elseif colorcheck >=4 and colorcheck<=7 then
                                agent.state=GREEN
                                agent:getCell().state=GREEN
                            else
                                agent.state=RED
                                agent:getCell().state=RED
                            end
                        colorcheck=0
                        end
                    end)
                end
            end)
        end
}
```

Code 14: Cultural transmission rule

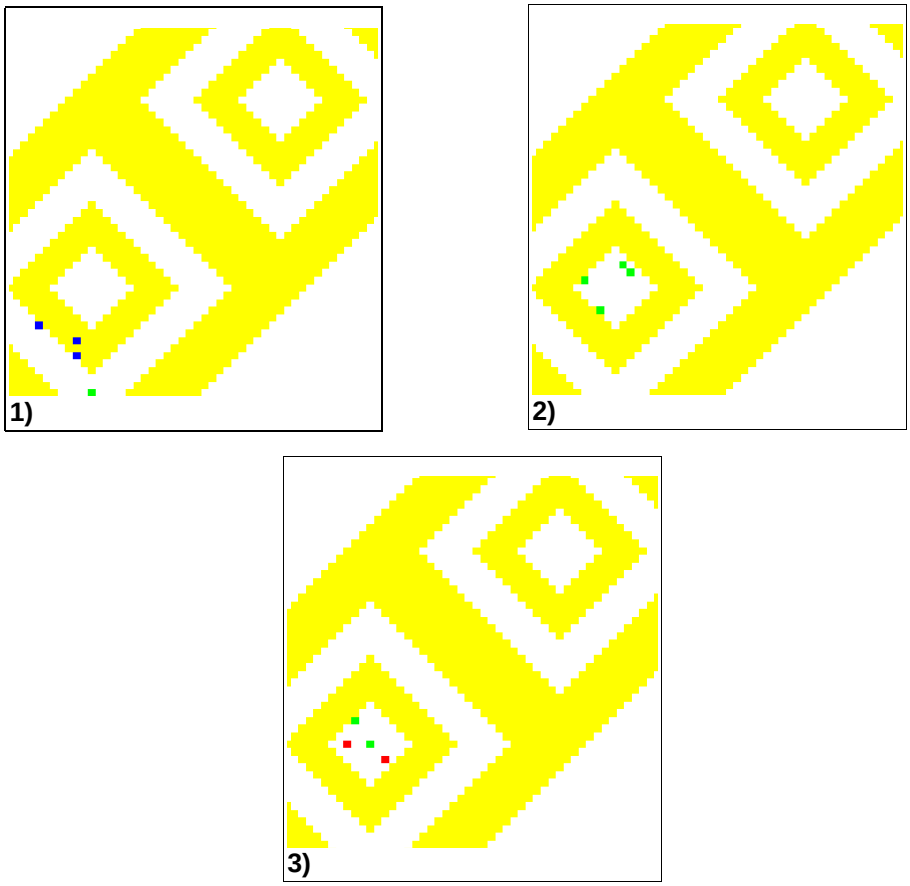


Figure 5: Agent cultural transmission behavior

3. Summary

As we can notice during this tutorial it is possible to achieve effective results with the help of quite simple behavior rules of the individual agents or the environment. This circumstance allows us to simulate scenarios with different initial settings that lead us from the micro level to the macro level. If we look at e.g. the spread of cultural tags. Are there any tribe grouping under certain conditions?
Furthermore we got to know types and functions that are adequate for ABM,
for setting up an environment from bottom up and to populate this environment. Also, we are able to control the initial states and the behavior of the environment and agents. This examples are just a starting point for creating a variety of scenarios within a society. So we can think of adding more scenarios like combat, more complex social behavior or trade to the model.

4. References

1. J.M. Epstein and R.Axtell (1996). *Growing Artificial Societies: Social Science From the Bottom Up*. MIT/Brookings Institution.

2. P. R. Andrade and T. G. S. Carneiro (2013). TerraME Types and Functions. Version 0.8. Available at www.terrame.org.

3. P. R. Andrade and T. G. S. Carneiro et al. (2012). Agent- Based Modeling in TerraME. Version 0.2.1. Publication unknown.

4. T. Carneiro and G. Camara (2007). A gentle introduction to TerraME. *INPE Report, Version 1.0*.

5. T. Carneiro et al.. (2013). An Extensible Toolbox for Modeling Nature-Society Interactions. Environmental Modelling & Software 46 (0): 104–117.

6. T. Carneiro and G. Camara (2013) “An Introduction to TerraME”. INPE and UFOP Report, version 1.8.

Internet (07.09.2013):

http://en.wikipedia.org/wiki/Agent-based_model

http://www.terrame.org