Programming assignment: Cellular Automata

1 Introduction

1.1 Cellular automata

A cellular automaton is a discrete model studied in mathematics, computer science, physics and biology. It consists of a regular grid of cells, which constitutes a space in which automaton events can take place [1] [2]. A cellular automaton system (or just a cellular system for short) is then constructed by formulating a set of rules governing the behaviour of the system in time. The grid can be set in any finite dimension and each cell is given a finite list of possible states. A transition function is further defined as a function assigning a state for each cell at the next time step based on the possible states and the rules governing the state transition [2]. The key idea is to formulate the rules in such a way that they only address the immediate neighbours of the cell and the cell itself. The study of a cellular space can be then brought to a much simpler (i.e. dealing with fewer dimensions) problem by considering the neighbourhood relations of each cell. Finally, a cellular automaton is uniquely established by the cellular system and its assigned initial state.

Setting up a problem as a cellular automaton gives rise to three fundamental features of such systems: locality, simplicity and parallelism [3]. The interaction in cellular automata are defined on the basis of locality — each cell interacts only with the neighbouring cells and itself. As a result, none of the cells have a global view of the system. The cellular automata simplify the problem under consideration because the basic unit of the automaton is the cell, which evolves in discrete time and space. The last feature, parallelism, means that the automaton performs computation on a spatially extended grid. The advantage of such approach is that the whole problem is not split into sub-problems and then combined to yield a final solution, but instead, the parallel processing is used [3].

1.2 Applications and relevance

The concept of cellular automata has been discovered in the 1940s and one of the most influential work has been done by Wolfram, who developed and proposed a uniform cellular automata construction [4] [3]. The work was a significant contribution to the field of computing and since then has led to a realization that simple rules can give rise to complex behaviour [5]. As a result, cellular automata have been used as idealizations of natural systems. It has been even suggested that they may model the nature’s phenomena better than traditional mathematical equations [1].

So far the cellular automata have been successfully used to represent a variety of phenomena. In physics, the concept has been applied to the diffusion of gaseous systems, solidification and crystal growth in solids and turbulent flows [1]. Cellular automata have proved highly useful for modeling and simulating systems which involve spatial arrangements, such as urban, social and economic systems. In biology, the model is the most known as a tool to explain the mechanism for pattern formation in natural systems. For instance, snowflakes or pigmentation found on mollusk shells show a great diversity in the complex pattern. The origins of such phenomena have been investigated, amongst other, through cellular automata [4].

Classical ecological models study the interaction between various species and heavily rely on cellular automata. The model enables researchers to define the interaction between populations with the desired simplicity of assumptions [5]. An original problem of Citrus Sudden Death (CSD) was tackled using the cellular automata. The idea was to develop a simple yet accurate methodology to study the disease which affected citrus trees in Brazil. Because the disease was thought to have been caused by a virus transmitted
by insects (aphids), one of the aphids’ enemy was selected to create a predator-prey type of model to study their interactions [1].

The approach to study how higher and lower levels of organization influence each other is particularly important in bioinformatics field [5]. Much focus is directed towards the problem of protein structure prediction and finding protein-coding and promoter regions [3], where the framework tool is based on the cellular automaton idea. An interesting application of cellular automata is also found in the pharmaceutical field. The engineering of polymeric biodegradable matrices, which effectively sustain and control the release of the drug and hence act as drug delivery systems, has required a combined effort and expertise. Cellular automata have been used as a tool to model the performance of these drug delivery systems and have been implemented in the drug design process [1].

Current mathematical models of natural phenomena are usually based on differential equations, which by definition give a description of a variation of one variable as a function of others. Cellular automata have proven to be a valuable alternative with a discrete evolution of (identical) components of a system as their output [4].

2 Results

2.1 The problem overview

In this project we consider a problem of a population of animals living on a cellular grid. Each cell of the automaton is in two possible states: 1 — representing the presence of an animal, and 0 — representing an absence of such. As a result, we assume that only one individual can be present at such location at one time. Additionally, we introduce another cell grid, making the cellular automaton double-leveled. The second grid represents food, which the animals can feed upon. Each cell in the food grid can be in one of the finite-numbered states, corresponding to a certain amount of food present in the cell.

The set-up of the problem enables us to introduce rules all the automaton cells follow.

1. An animal grazes in within one cell distance from its position.
2. The grazing areas of animals can overlap.
3. Animals require a minimum amount of food to survive. If the amount of food drops below a critical value, the animal dies (disappears from the grid).
4. An animal eating food reduces the amount of food.
5. Food regrows at a constant rate.
6. An animal can breed one cell up, down, left or right provided that the cell is free (in state 0) and there is enough food in the corresponding cell on the food grid.

These neighborhood relation and transition function are the same over the whole space, except for the edges of the grids. Animals cannot breed or graze in such locations. The cells with these locations are hence permanently empty (in state 0) or with a constant amount of food, for the food and animal grid, respectively.

We also define one generation as an iteration of operations of the grids, so that in one generation an animal eats and breeds only once (in that order). All four directions of reproducing are equally probable to be chosen, but an animal can only successfully reproduce if the randomly chosen cell is empty and contains enough food for one animal. Having established the rules governing the grids, the animal population is allowed to live during $n$ iterations (generations). The population's behaviour, as well as resulting food supplies, are displayed graphically.
2.2 The code description

The code starts by prompting the user to choose initial variables determining the initial state of the system. The user is asked to specify the length and width of the grid where animals live, as well as the initial number and positions of the animals. "If" statements serve as checkpoints to determine whether the values chosen are in the correct range. If they are not, the code overwrites the wrongly chosen values to the closest possible one (see Fig. 2). Having checked them, an initial animal grid is printed out (see Fig 1b).

A NumPy package was chosen to deal with the arrays due to the relatively simpler manipulation of arrays and some useful functions already included in the package. The arrays are defined with a size bigger by two to account for the treatment of the edge effects. The operation does not affect the ease of choosing the positions of animals, because the cell arrays are indexed from a zero value and the positions from the number one.

The next part initiates the process of grazing and breeding of the animals put on the grid by the user. Another prompt asks for a number of iterations (i.e. generations) to be calculated. Then, two nested 'for' loops look for cells with animals in them. If any are found, the code creates two temporal three by three arrays to manipulate the local environment of the animal. An appropriate amount of food is subtracted from the temporal food array (provided

---

Figure 1: a) The flowchart of the code. b) An example of the output of the code in the terminal.

A NumPy package was chosen to deal with the arrays due to the relatively simpler manipulation of arrays and some useful functions already included in the package. The arrays are defined with a size bigger by two to account for the treatment of the edge effects. The operation does not affect the ease of choosing the positions of animals, because the cell arrays are indexed from a zero value and the positions from the number one.

The next part initiates the process of grazing and breeding of the animals put on the grid by the user. Another prompt asks for a number of iterations (i.e. generations) to be calculated. Then, two nested 'for' loops look for cells with animals in them. If any are found, the code creates two temporal three by three arrays to manipulate the local environment of the animal. An appropriate amount of food is subtracted from the temporal food array (provided

---

Figure 2: An example of the output of the code in the terminal when the input variables are out of the range. The code overwrites the inappropriate values and confirms when an animal is successfully added onto the grid.

---

Pick an integer length (x axis) of the field: 3
Pick an integer width (y axis) of the field: 3
How many animals would you like to place on the field? 3
The x position of the animal number 1
The y position of the animal number 1
An animal number 1 added successfully!

Pick an integer length (x axis) of the field: 6
Pick an integer width (y axis) of the field: 6
How many animals would you like to place on the field? 3
The number of animals can’t be larger than the number of cells!

The number was overwritten to the maximum possible: 0.
The x position of the animal number 1
The position is outside the field
Overwriting to the nearest position: 3
The y position of the animal number 1
The position is not an index - cannot be 0
Overwriting to the nearest position: 1
Animal number 1 added successfully!
there is enough food in the cells). Thanks to the NumPy package, any changes made to the temporal array are also updated on the main food array.

Next, an integer number is generated to pick a random direction for the animal to breed. There are four possible directions for an animal to breed, and, if the randomly chosen cell is free and there is enough food in it, an animal reproduces.

The second part of the generation loop deals with the additional rule imposed on the grids. An argwhere NumPy function returns indices of the cells where a specified condition is satisfied, which enables a quick change of the state of the cells. First, if the amount of food in a cell is less than what one animal eats in one-time step, the animal is killed (i.e. the state of the cell is changed from 1 to 0). Food is added to all the cells of the food grid according to the specified food growth rate. If any cells of the food grid end up with a negative value or a value over 255 of food, the code overwrites the current value to 0 and 255, respectively. This ensures that the grid can be easily visualized in a gray-scale image. Finally, the borders of the arrays are updated to their initial values, that is zero and 128 for the animal and food array, respectively. This effectively restricts the animals from living on the edges of the array.

After each completed iteration both the food and the animal are displayed in the terminal. If there are no animals present on the grid an information about the extinction of the population is printed out.

In the final part of the code, two files are created and the code writes the information in the Netpbm format. The arrays are written to the file after being translated back to the normal array format. The files are then closed and saved in the current directory (see an example in the Fig 3).

**Figure 3:** An example of the output of the code in the form of pictures automatically saved to the current directory. Two different sets of rates were compared here. First two columns correspond to the food growth rate higher than the rate at which the animals eat, while the last two to the opposite situation. In both cases the initial conditions of the grid were the same — the grids are of the same size, the numbers of animals and their positions are identical. The population consuming food faster then it can grow is extinct after two generations.

3 Discussion

In the presented code, the neighbourhood relations and the transition function are the same over the whole grid (except the edges). Further implementations could allow variations from cell to cell, provided that the relations would still be calculable from the coordinates of the cell. Such changes might include making the food grid resemble a particular environment, in both spatial and time sense.
Periodic conditions could be imposed on the edges of the grid, as they are defined to be unpopulated, which is an obvious limitation. However, such a solution requires the use of a toroidal array, beyond the scope of the Programming course (and time allocation for the project). Alternatively, the cellular automaton could be allowed to expand in size, so that dynamic storage allocation could be used to hold the growing patterns.

Studying interactions and patterns over large scales is of particular importance for ecosystems. The interactions in our problem are assumed to be local, and so they are in real systems [5]. Studying large-scale patterns gives an insight into how large scales influence the dynamics of the system as a whole.

Finally, an interesting notion of evolution or mutation could be applied. In nature, cells and their properties evolve through transcription regulation, mutations, stress or strain. The cellular automaton could be allowed to spontaneously expand or change the prior defined set of states and their transition rules imitating the evolution influence. The rules that govern living cells are not arbitrary and not necessarily as simple as possible. In fact, any living system has a high design of freedom [6] and as a result could, in theory, choose a solution quite different from the assumed simple solution. Therefore, an evolutionary perspective is necessary for understanding living organisms [5].

We have significantly broadened our knowledge about the components in biological and physical systems in recent years, yet we still know little about the mechanisms between the components which give rise to the observed complexity. A mathematical theory is sought that could describe the natural phenomena by generalizing them. Analyses of cellular automata are proving helpful in developing models for specific systems. Starting with a relatively simple approach enables us to connect it to a complex behaviour. This, in turn, one would hope, could lead to mapping complex systems to each other and extracting general principles that could be applied to a variety of systems.

References


The code
# /usr/bin/env python
import numpy as np
import random

# Define some variables:
Food_Level_I=128  # The food grid visualized in grayscale (0-255).
Food_Regrowth_Rate=15
Food_Eaten_Rate=30

# Ask for the size of the field:
print "Pick an integer length (x axis) of the field: 
Field_Size_x=int(raw_input())
print "Pick an integer width (y axis) of the field: 
Field_Size_y=int(raw_input())

# Create arrays of an appropriate size:
Food_Array=[]
Animals_Array=[]
Animals_Array=np.zeros((Field_Size_y+2, Field_Size_x+2), dtype=int)
Food_Array=Food_Level_I*np.ones((Field_Size_y+2, Field_Size_x+2), dtype=int)
#+2 to create a grid edge which account for boundary effects

# The initial set of animals:
No_Animals=0
print "How many animals would you like to place on the field?"
No_Animals = int(raw_input())
if No_Animals == 0:
    print "You have to place at least one animal.\n" # Each input value is checked to be make sense.
    No_Animals=1 # If not, it's overwritten to a closest value
if No_Animals > Field_Size_y*Field_Size_x:
    print "The number of animals can't be larger than the number of cells.\n" print "The number was overwritten to the maximum possible: %d \%(Field_Size_y*Field_Size_x)
No_Animals = Field_Size_y*Field_Size_x
for n in range(1, No_Animals+1):
    print "The x position of the animal number %d" %n
    xPos=int(raw_input())
    if xPos > Field_Size_x:
        print "The position is outside the field"
        xPos = Field_Size_x
        print "Overwriting to the nearest position: %d" %Field_Size_x
    if xPos == 0:
        print "The position is not an index - cannot be 0"
        xPos=1
        print "Overwriting to the nearest position"
    print "The y position of the animal number %d" %n
    yPos=int(raw_input())
    if yPos > Field_Size_y:
        print "The position is outside the field.\n"
        yPos = Field_Size_y
        print "Overwriting to the nearest position: %d" %Field_Size_y
    if yPos == 0:
        print "The position is not an index - cannot be 0"
        yPos=1
        print "Overwriting to the nearest position"
    if Animals_Array[xPos][yPos]!=0:
        print "There can only be one animal in each cell.\n"
        print "The animal number %d could not be added.\n" %n
    if Animals_Array[xPos][yPos]==0:
        # An animal can only be placed on a grid
        Animals_Array[xPos][yPos]=1 # if the cell is free.
        print "Animal number %d added successfully!" %n

print "You have placed your animal(s) here:\n"
print Animals_Array
print "-----------------------------"

# Some variables needed for this part of the code:
FTemp=[]
ATemp=[]
RandomRepr=0
tmax=0

# t represent a single time step in which an animal can eat and reproduce (i.e. generation)
# Ask about the number of iterations to be computed:
print "How many next generations would you like to see?"
tmax=int(raw_input())
if tmax==0:
    print "Number of iterations should be non zero.\n"
    tmax = 1
    print "The number was overwritten to the smallest possible: 1.\n"
    print "--------------------------------------\n    "
    print "Iterations start now"
    print "--------------------------------------\n"
for t in range(1,tmax+1):
    Iterate through each element of the animal array
    for i in range(1, Field_Size_y):
        for j in range(1, Field_Size_x):
            # If an animal is found:
            if Animals_Array[i][j] == 1:
                # Create a (temporal) local food array:
                FTemp=Food_Array[(i-1):(i+2),(j-1):(j+2)]
                # Return indices where there is enough food to be eaten:
                Rule_Food = np.argwhere(FTemp > Food_Eaten_Rate)
                # Animal can eat from these cells:
                FTemp[Rule_Food] -= Food_Eaten_Rate

                # Create a (temporal) local animal array:
                ATemp=Animals_Array[(i-1):(i+2),(j-1):(j+2)]
                # Generate a random int from a set (1,2,3,4):
                RandomRepr=random.randint(1,4)
                print "Random reproduction direction: %d" %RandomRepr
                # Each random integer corresponds to one direction an animal can
                # breed in. An animal reproduces provided the chosen cell is not
                # already occupied and there is enough food in it.
                if RandomRepr==1 and FTemp[0][1]>Food_Eaten_Rate and ATemp[0][1]==0:
                    ATemp[0][1]=1
                if RandomRepr==2 and FTemp[1][0]>Food_Eaten_Rate and ATemp[1][0]==0:
                    ATemp[1][0]=1
                if RandomRepr==3 and FTemp[1][2]>Food_Eaten_Rate and ATemp[1][2]==0:
                    ATemp[1][2]=1
                if RandomRepr==4 and FTemp[2][1]>Food_Eaten_Rate and ATemp[2][1]==0:
                    ATemp[2][1]=1

                # Impose the rest of the rules:
                # Return indices of the cells, where there is not enough food.
                Rule_Food_starve=np.argwhere(Food_Array < Food_Eaten_Rate)
                # Animals in these cells die.
                Animals_Array[Rule_Food_starve]=0

                # Return indices of the cells, which have a negative amount of food.
                Rule_Food_min=np.argwhere(Food_Array<0)
                # Amount of food can't be negative, so overwrite to 0.
                Food_Array[Rule_Food_min]=0

                # Food regrows everywhere.
                Food_Array+=Food_Regrowth_Rate

                # Returns indices of the cells, which have more food than 255.
                # 255 is the maximum amount of food that can visualized in grayscale.
                Rule_Food_max = np.argwhere(Food_Array > 255)
                # For these cells, overwrite the amount of food to the max value (255).
                Food_Array[Rule_Food_max]=255

                # Ensure the boarders on the animals' grid stay null.
                # Animals can't breed on the boarders.
                Animals_Array[0, :]= Animals_Array[Field_Size_y+1:, :]
                Animals_Array[:, 0] = Animals_Array[:, Field_Size_x+1] = 0

                # Ensure the boarders on the food's array stay at the initial declared level.
                # Animals can't eat from the boarders.
                Food_Array[0, :]= Food_Array[Field_Size_y+1:, :]= Food_Array[:, 0]
                Food_Array[:, Field_Size_x+1] = Food_Level_I

            # Print the resulting grids after each iteration.
            print "The food after %d iteration(s)" %t
            print Food_Array
            print "--------------------------------------\n            "
            print "The animals after %d iteration(s)" %t
            # If there are no animals left on the grid, print it out.
            if np.all(Animals_Array[:,0]==0)==True:
                print "The population is extinct."
print Animals_Array
print "--------------------------------------"

# This part of the code saves the food and animal arrays into a file as images:
tname=str(t)
resol=str(Field_Size_x+2)+" "+str(Field_Size_y+2)
FoodAsList=[]
# Save the arrays as normal lists and invert the grayscale (for the food array)
# so that darker colours correspond to more food.
FoodAsList=np.asarray(255-Food_Array)
AnimalsAsList=np.asarray(Animals_Array)

# Open the file if it exists or create if it doesn't.
fanimals=open("animals"+tname+".pbm", "w+")
stringline = " "
commentanimals = "# animals"+tname+".pbm\n"
fanimals.writelines("P1\n")
fanimals.writelines(commentanimals)
fanimals.writelines(resol+"\n")
# Write each row in the array as a string, because the write
# option requires a string or a buffer.
for line in AnimalsAsList:
    stringline=" ".join(str(e) for e in line)
    fanimals.writelines(["%s\n" % stringline])
# Close the file.
fanimals.close()

# Similarly for the food array.
# The only difference is the different format because of the grayscale.
ffood=open("food"+tname+".pgm", "w+")
stringline = " "
commentfood = "# food"+tname+".pgm\n"
ffood.writelines("P2\n")
ffood.writelines(commentfood)
ffood.writelines(resol+"\n")
ffood.writelines("255\n")
for line in FoodAsList:
    stringline=" ".join(str(e) for e in line)
    ffood.writelines(["%s\n" % stringline])
ffood.close()