Cell Magician: final report

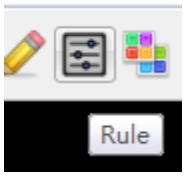
Xiang Sitao, Wu Zhelun, Wan Jun
Instructed by Hua Qiangsheng

1. Basic introduction of the cellular automaton

   Cellular automaton, devised by the famous John von Neumann, was first presented as an attempt to simulate cells' behavior on a computer. It is a spontaneous program, requires only initialization and no further input. One interacts with the cellular automaton by creating an initial configuration (starting mode), an initial series of rules for how the cells perform and then he could observe the cells' evolutions.

2. Function Introduction and User's Guidance

   (1) Create a user defined rule

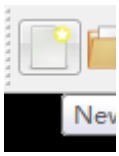
   

   Users can first select a rule to their own preference. For each rule, we allow users to give reasonable parameters, for example, the number of living cells around to determine whether should we create, keep or mop out the central cell (Life Like), so that different kinds of rules would be available.

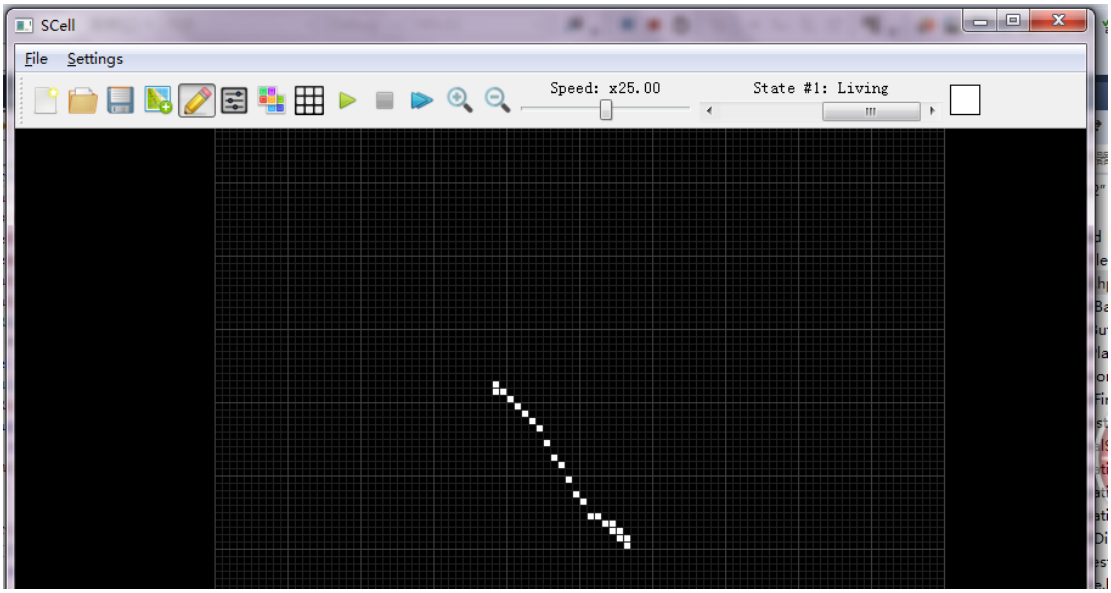
   (2) Change the size of the total grid

   For users' conveniences, we allow users to adjust the cell's size through sliding the wheels up and down. When the cells' size is small, we can observe the whole grid's pattern, while in the other case, we can observe the shifting of each specific cell.

   (3) Give initialization to the grid

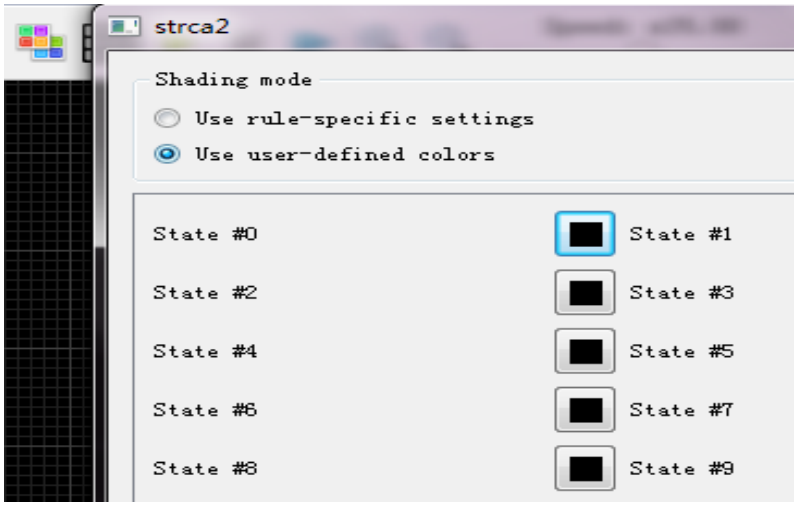
   

   To start the Cell Magician, we will first have to start at some pattern. We allow the user to color cells by clicking the left button of the mouse while discoloring cells by clicking the



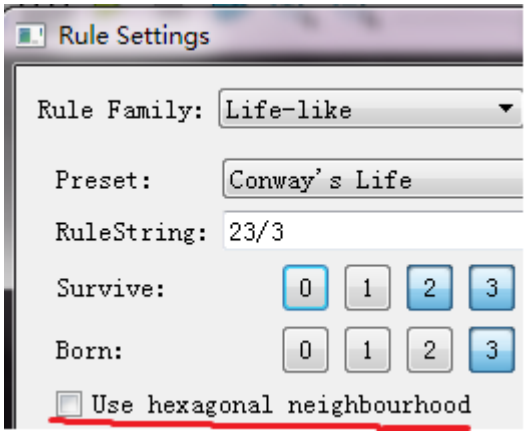
right button of the mouse. Users can also have randomized initialization by setting the percentage of each state shown on the board. Besides, the initial setting, users can color or discolor anytime they like by clicking the stop button, and repaint.

(4) Select color for each state



We allow the users to pick their favorite colors on the menu, so as to maximize the visual effect of our patterns.

(5) Choose the cells' shape



We offer two different kinds of cells' shape, namely, square and hexagon. Users can click the rule setting button or enter through the menu, you would see there is an option saying if you want to use hexagonal neighborhood, and you will get hexagonal grid after you

confirm it.

(6) Control the evolution process at will

The users can start the running process by clicking our start button, the space key or setting step number as you want by clicking go-to generation button and enter the number.

While running the algorithm, users can halt anytime they like by pushing the pause button.

(7) Speed up the calculation

On the top of our user's interface, there is a slide bar representing the changing speed of the pattern; dragging the slide block will change the speed of evolution.

(8) Store and read a preferred pattern into a file

Click the Save button and we could store the current pattern into a file, which is readable only for program, not for human. When restarting the program, one can click open the stored file to regain the pattern.

(9) Store and read a rule into a file

We allow the users to store his/her favorite rules into files, so that it can be reused later. This part of the code is finished, but not yet inserted into the program, because we are still not sure how the file's format should be like.

3. Implementing details

(1) Rule Introduction

There are totally seven kinds of rules for the users to choose, namely Life Like, Generation, Forest Fire, Gas Test, Wire World, Margolus and Reaction Diffusion. In all the above rules, we allow users to give initial parameters to their own liking. For example, in the Forest Fire rule, users can set the burning rate and growth rate in a reasonable range to create distinct or even maybe completely different visual patterns.

The most basic rules are probably Life Like and Generation, which take a cell's nine surrounding neighbors' states as input and thus decide a cell's next state. The cell's next state relies only on the number of the surrounding neighbors' living cells and nothing else.

More enhanced rules focus on application in real life. For example, Forest Fire can simulate the burning and growing of a forest; Gas Test can simulate the atomic movement of gas; Wire World can simulate electrons running in a circuit; Margulas can simulate the crystallization of particles, or even the process of sands or water falling down under the influence of gravity. They are all very beautiful. For detailed knowledge of how this rules work, please see our project attached.

Thirdly, we would like to mention the Reaction Diffusion rule, which

takes inspiration from Professor Alan Garfinkel's lecture on "Turing's Work on Pattern Formation". In our rule, we can set different parameters to simulate the combination, formation and interaction of different pattern, detailed use of which can be seen in our project.

Also, to offer the readers with more convenience, we provide a way to store a rule into a file, which later can be read to recreate that rule. This function is also useful in the case that some beautiful rules might be highly irregular and may require large number of space. To increase the running speed of our project, we decided that rules with more than 12 states will be stored in the file format, so instead of creating a rule based on that file, we simply visit that file every time to fulfill the calc() function.

(2) Rule Implementation

As rules all share a lot of similar properties, we decide to create a virtual base class called RuleBase, in which we record the basic functions for all rules, like int stateCount(); int period() etc.. We then create three more bases to differ rules in three different classes, based on whether their next states depend on only their neighbors (it can also depends on the running time, the neighbor's neighbor etc.).

So in implementing each rule, we first select the base class that we would like to inherit from, and then specify the implementing details.

Now we would like to introduce the basic function for a rule:

vector<string> stateNameStr: records the name for each state so as to offer the readers convenience in understanding the pattern.

string rulename: briefly introduce the initial parameter user offered for a certain rule

unsigned char table[2][9]: recorder of the next state

int m_stateCount: used to record the maximum possible number of state;

int period(): the changing period of a certain rule

int stateCount(): return the m_stateCount;

int shape(): return if the present grid used is square or hex;

bool needConversion(): if the whole grid might require conversion in a move;

unsigned char calc(unsigned char state1[], unsigned char state2[], unsigned char state3[]): calculate the next state of the cell;

void Setrule(char** nRule, int n_stateCount): used to setrule to Lifelike;

string stateName(unsigned char state): return the name of a certain state;

string ruleName(): return the rulename;

For the implementing details of the rule class, please refer to our project file.

(3) Grid Introduction

In our project, the map is designed to be very large, so it will be extremely inefficient to take care of the whole grid simultaneously. So instead, we cut

the whole grids into 256*256-sized regions, while creating a class Grid to manage the evolution process of every region, deleting empty ones and creating newly-borns.

In order to increase the efficiency of the Grid, we will have to introduce a binary search tree to keep all regions in order. After discussion, we decide to use a Red-Black Tree, as it can retrieve the region wanted in a rather short period of time while saving large amount of storage space. We can only store regions with non-empty cells, and in the meantime, keep them in order by sorting their coordinates as keywords. As it turns out to be, this is highly effective, with a running time much faster than that using a linked list (which we also implemented).

Also, Grid class greatly reduce the burden of the Board implementation, as it dealt with the problem of regions aligning and dissipation, rather than leaving it to the later drawing part.

(4) Grid Implementation

The Grid class contains information about the current minimum and maximum in x and y in cell's coordinates on the original map, the minimum and maximum in x and y in region's coordinates, the generations passed since the first run, and of course, the root and sentry (NULL node) of the red-black tree. In the implementation, red-black tree node is separated from the Grid (because in each Grid we only need two red-black tree nodes), which means that we must employ 'friend' template function to visit the member in the Grid class when implement the tree node removal and insertion. stat[] array is used to store the status of the 256 cells in each region to indicate whether they are old or young or some more general status they are in.

Now we would like to introduce the basic function for a Grid class:

Region<ValType> *findRegion(int rx, int ry):find the region with the region location (rx,ry);

Region<ValType> *getRegion(int rx, int ry): find the region with the region location (rx,ry) and if there is no such region, create one;

void randomize(double *pos): randomize all the regions;

void setRule(Rule<ValType> *p_rule): set new rule to each region;

void evolve(): evolve each region;

void clear(): clear all the region in the Grid and be ready for the new evolution;

void setup(int w, int h): initialize the Grid with a certain width and height;

void putCell(int x, int y, ValType state): put a certain cell on the board map;

inline Rule<ValType> *getRule(): return the current rule;

inline long long getTime(): return the current time(generation number);

ValType getCell(int x, int y): obtain a certain cell in the region;

long long population(): return how many cells are alive on the board map;

long long population(int index): return how many cells are on the board map of the status index;

(5) Region Introduction

As mentioned above, the whole grid is divided into 256x256 regions. The Region class handles the detailed control of evolution process in such a local area.

To speed up computation, this class uses a carefully designed method to tell which cells need to be updated and which not, reducing total amount of computation.

To handle the border between regions more efficiently, objects of Region class are connected into a matrix by 8-direction doubly-linked lists.

(6) Region Implementation

In the implementation of Region class, the 256x256 area is further divided into chunks of size 16x16. Basically, for each chunk, we keep a record of whether this chunk has any of its cell changed in the last generation. If so, we consider this chunk necessary to be checked for update in the next generation. If not, we expect that in the next generation, this chunk will still remain unchanged, so we don't need to consider it. This is done in function evolve() by simply setting s flag during evolution process.

The problem is that if a cell on the border of a chunk changes state, it might case a change in neighboring chunks during the next generation. So we designed a function cleanUp() to propagate the update flag from each chunk to its neighbors. Besides, this function also copies cells on the border of the region to neighboring regions.

(7) Board

The board is the main part of our user's interface, which receives signals from the keyboard and the mouse, based on which do operation on the grid and shader. We realize the zooming in and out (size-shifting of the board) through a certain scale factor. We also realize both hexagonal and squarish grid, whose mechanisms are completely different, including the conversion between the cell location we defined, the real pixel we are about to paint on our board and how should we propagate. After many difficulties in dealing with the hexagonal grid problem and we come up with the idea that we could draw three series of lines that are parallel to the three pairs of the hexagon and use these new lines to determine the location of the cell. On the other hand, since there could be different status to each cell, especially between the diffusion and the ordinary rule, we have a BoardBase abstract class to inherit from. The abstract class has its virtual functions and its own signals and slots, and we specifically define all those in the Board class later so that we could deal with different cell records. The way we choose to paint depend on the Boolean variable which indicate whether we are about to paint a single cell, part of the rectangle or the whole board.

Now take a look on the basic function of the Board class:

void pixel2Cell(int px, int py, int &cx, int &cy): convert the pixel location

into the cell location of the cell which contains that pixel;

void cell2Pixel(int cx, int cy, int &px, int &py):return the pixel location at the lowerleft of that cell (cx,cy);

void pixel2CellHex(int px, int py, int &cx, int &cy): convert the pixel location into the hexagonal cell location of the cell which contains that pixel;

void cell2PixelHex(int cx, int cy, int &px, int &py): return the pixel location at the upperleft of the bounding rectangle of the cell (cx,cy);

void putCell(int px, int py): by using the Grid class's putCell, we store the change to the Grid and repaint the whole board and also emit a signal to change the population;

void updateZoom(): when we change the scale factor, we are about to repaint the graph and each time we change the scale factor we have to change the corresponding width and height;

void renderControl(): with render control we set those Boolean variable and the parameters we need for furthering painting in the paintevent;

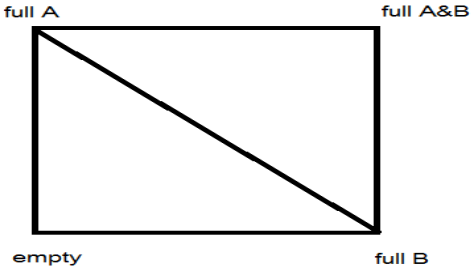
(8) Shader

Shader here is used to determine the color we are about to draw on the panel and it is included in the RuleSetting class and the RuleSetting class associated with the name of each state, or in other words, a number with a certain color.

In each pre-set rule, we have our own default shader, which can be used to associate the type to concrete color and make it easy for us to adjust the color to users' favorite color. The default shader is very well designed so that we could have a direct understanding how cells have evolved or how particles spread around.

In the generation shader, for example, we separate the color by dividing the color's three parameters, red, green, blue value evenly from the living cell to dead cell which is black, so we could see that the color will gradually turns from yellow to orange to dark orange.

In the reaction-diffusion shader, however, we divide the mixture of two chemicals according to their percentage in the following way. We first have the basic four color of states which represents the empty, full A chemical, full B chemical and both A&B chemical are full. and we make them as the four vertices of the square, like the coordinates, and generate color in two part and if the chemical concentration is in the left lower triangle we based the color parameter on the three vertices of the left lower triangle by assigning proper weight and similarly when the concentration falls on the upper right triangle.



For implementing details, please see out project enclosed.

4. Other's implementation

(1) Dialog and Panel in the Users' Interface

We create different dialogs including setting rules, showing Grid, and also controlling the shaders so that the users can set parameter to initialize our starting configurations.

(2) Random Initialization

Since we have divided the board into regions, we can also randomize the whole board in each regions and according to the knowledge of probability, the expectation number of cells agree with the total expectation percentage we assigned. This is accomplished by an algorithm based on the rand() function.

(3) Process Control

In order to make our program react to user's implementation instantaneously, we emit signals every time when there should be a stop. It could either appear in the 'space' press where we implement 1 generation at a time or we press the stop button in the tool bar. And we allow users to change the current state by using the pen to draw any status of cells on the board and see the following evolution. We even allow rules to be changed during the evolution without resetting the whole board to blank in order to better simulate the real evolution when climates and environment are likely to change.

(4) File Store/Read

To store a rule, we merely have to store its table[][], plus some basic information like m_stateCount. But table[][] can be very large, so we use a special format to store the table[][], for more detailed information about this format, please refer to the Data Structure part.

To store a grid, we use LZW to compress it, so that large space might be saved. When compressing a grid, we first find out all its living regions, each region will then be recorded using LZW. In this case, the file size is acceptable.

To read a grid-stored file, we will merely have to decompress it and create a new grid correspondingly.

If we want to construct a rule based on the stored file, then we will have to use a large perfect hash table to replace the original table, the detailed coding process can be seen in the Data Structure part.

Else if the state number is too large, then we can only read the file for the calc() function. In this time, we use a method similar to that of the digital search tree. We first find out the center cell's state, and the lines in the fill correspondingly. Then we find out the number of state0 in the neighbors and further narrowing the range of lines. So using this process recursively, we will eventually find out the new state.

5. Data structures used

(1) Red-black tree

Red-black tree is the basic mechanism when we deal with the enormous amount of regions in the Grid. The reason why we choose this data structure is that we want to divide-and-conquer, to make the whole evolution separated into different regions so that we could deal each part easier. And red-black tree is very efficient when a lot of insertion and deletion are executed. Moreover, we could save the space because we only need to store those region where there is at least one cell is alive.

(2) Digital Search Tree

Digital Search Tree is used in correspondence with the LZW Compression, to store the dictionary presently available in the LZW process. The reason we choose to use it is that it has small height compared to other priority queue, and that we can access the data with ease.

(3) Lempel-Ziv-Welch Compression

As our grid can be very large, so we will need an efficient compressing algorithm to store the state of each cell. After discussion, we decide that LZW is the most appropriate way. Sometimes, a 16*16-sized chunk can be recorded using only ten characters.

In the coding process, we use DST, as we can finished searching a number in O(log n) time. However, in the decoding process, we doesn't need to search the dictionary anymore, so we simply use array, which can access any given number in O(1) time.

(4) Rule Storing Format

We first show an example of the Wire World's stored file:

```
// WireWorld
// Simulate electric circuits
rulename=WireWorld
ruleclass=Outer Totalistic
statecount=4
neighbour=0,1,2,3,4,5,6,7
statename0=Empty
statename1=Wire
statename2=Electron Head
statename3=Electron Tail
table:
s=2 >3
s=3 >1
s=1 #2=2,3 >2
```

Note that we first record the basic information like the m_stateCount, period etc. then the table at last. We first show the center cell's state, then the number of state0, state1, state2, ---- in its neighbors, and lastly >3 would imply that the center cell goes to state 3 in this case. For example, s=2 >3 implies that if the center cell's state is 2, then its next state is 3. If we can not find a state in the file, for example, s=0, this means that its next

state is the same as before.

(5) Hash Table

To transform a file back into a rule's table, we will use a large hash table to record each state, for example, if a cell is in state 1, with all neighbors in state 2, then the corresponding key is 1 * statecount ^ 9 + 2 * statecount^ 7. In this case, we can guarantee that there should be no collision, and that eventually it only take O(1) to code, decode and access new state.

(6) Multi-linked list

Besides, the Red-Black Tree structure in the grid, we have also implemented the grid using a multi-linked list. That is any region is connected with the 8 regions in its surrounding so as to access their data easily. This data structure offered much convenience in coding but is rather slow in running. So in the end, we decide to use the Red-Black Tree to store the grid.

(7) Conclusion

In the programming process, we have tried different data structures, now we would like to give a conclusion to our exploration.

LZW process: digital search tree is much better, with running time 5times smaller than that using array;

De_LZW process: array is better than digital search tree;

Grid implementation: Red-Black tree is better than multi-linked list;

Rule-storage and extract: hash function is convenient and efficient, and that sometimes, special file-format has to be created to enhance the efficiency.

6. Problems encountered and their solutions

(1) Problems with Coding Technique

At first, the lzw process is very slow, requiring more than 3 minutes for even a 256*256 region. And as later turned out, this arouse because of terrible coding technique. For example, lines like "vector.erase(vector.begin());" are frequently used, which making the algorithm jumping to O(n^2) level. So after a clear search, the program becomes fine and can finish storing a region in less than 1 second.

(2) Problems with cooperation

As the three of us use completely different system: mac OS, Windows and Linux. So it becomes at first difficult for code to merge together and program may crashes under different operating system while works on the other, but this on the other hand help us to find the faults in our program. But as C++ and QT are quite universal, we each set our environmental path into corresponding value and eventually overcome this obstacle.

(3) Memory leaks

When running the lzw process, the program forgets to disallocate used memory, causing Xiang Sitao's computer to stuck for nearly half an hour.

But it become alright after ~DST() are redefined.

Memory leak also happened in the Grid class when forgetting to delete the region inside, which led to mistakes in executing the program.

(4) Problems with Grid class

Indeed, Grid class have faced numerous mistakes since the time it was written. With many considerations, the Grid class has to be separated from the RB tree node, since what Grid really need is the member function which could help the board to manage each region. And with the class template and the intertwined call between the Grid class and RB tree node class, we should set a lot of friend class and friend global functions to help with the recursive node-finding and tree node removal. Also, the program used to crash very often because we do not follow the real tree root in the Grid class, so after discussion, we pass the Grid instance into each function parameter and this helps greatly with the stability of the Grid class.

Also, there could be regions being inserted when we cleanup the region, so we have to store them in another list at the same time to ensure that we will not evolve the same region for more than one times. Otherwise, during the rotations, we may make our tree-traversal in a mess.

7. Works dissipation
   (1) Data structures, basic ideas, proposal and report are discussed and realized by all three together.
   (2) Wan Jun implements the rule part, Xiang Sitao implements the region part and Wu Zhelun implements the grid part.
   (3) The QT's structure is created and mostly completed by Xiang Sitao, while Wu Zhelun helps in creating dialogs and Wan Jun helps in implementing some rule-related part.
   (4) The file part is written by Wan Jun, with major help from Xiang Sitao.

8. Acknowledgment

Thanks for out instructor Qiangsheng Hua, and the two T.A. Hu Li and Le Zhang for their generous helps. We would also express in here, our gratitude for all our classmates, whose very existence give us inspirations.