

Forum:[Hack Forums](#)

Thread: [Fun Project: Elementary Cellular Automata](#)

Kantos - January 16, 2017, 8:19 pm

Greetings,

I'm just posting a fun project for anyone here to do.

Elementary cellular automata are pretty simple. You start with a one-dimensional array of cells that each have a binary state, either "on" or "off", which we'll call 1 and 0 respectively from now on. You calculate the next state of each cell by looking at its two neighbors and following one of 256 possible rules.

If I show an example, it will make more sense:

This image shows Rule 90, which generates a Sierpinski triangle given a starting state of a single cell set to 1 and the rest 0. Time increases down the vertical axis, i.e. each row represents the previous row after applying the rule.

Rules are generally represented in this form:

Code current pattern	111	110	101	100	011	010	001	000
new state for center cell	0	1	0	1	1	0	1	0

As you can see, the second row is the binary representation of 90, hence why this is called Rule 90 and why there are 256 possible rules.

If you look carefully, you can also see that Rule 90 is the same as XORing the two neighbors (1 if they are different, 0 if they are the same).

Other popular rules include:

Rule 110 (proven to be Turing-complete): https://en.wikipedia.org/wiki/Rule_110

Rule 30 (found in nature, produces seemingly random patterns):

https://en.wikipedia.org/wiki/Rule_30

Rule 184 (can be used to model traffic flow, particle deposition, and ballistic annihilation):

https://en.wikipedia.org/wiki/Rule_184

The Wolfram MathWorld article shows a sample of every single rule:

<http://mathworld.wolfram.com/ElementaryCellularAutomaton.html>

Stephen Wolfram was actually one of the pioneers in this field.

I find these and all cellular automata to be very interesting, how such patterns and chaos and complexity can be achieving using only a very simple ruleset. It's a nice toy program to play around with, just like with Conway's Game of Life.

So, my project to you is: Write a program that implements some or all of these rules. Implementing one rule takes less than 50 lines, and implementing all of them can also be done in less than 50 lines. I have a function `rule()` that takes the triple pattern and an unsigned char with the rule number and returns the new state.

You don't have to do it in under 50 lines though, mine is very barebones (changing the rule or starting config or size requires recompilation). Also, I use spaces instead of zeroes.

Annabelle Lee - January 17, 2017, 7:05 am

This looks rather interesting, I'm a bit curious about the edge case however. Since I'm not sure how the rule states for the edge class when you only have one neighbor. Here is an example of what I mean:

```
Code Box# [1] [2] [3] [4] [5] [6] [7] [8]
Init 0  1  0  0  0  0  0  0
```

So take a look at slot #1, for the next generation, the value would be the XOR of the neighbours, in this case it would be the XOR of 2... and what else? If there is no neighbour, do I XOR it with a 0 or a 1? If you could please clear that up I can finish working on the project.

Kantos - January 17, 2017, 9:09 am

Oh yeah, technically there isn't supposed to be an edge case since mathematically it's an infinite array. I just hold the two ends constant, i.e. I only change cells 1 through n-2 (0 and n-1 being the ends). For simulations that "grow" like Rule 90 generally you stop the simulation before it reaches the bounds so it's fine. For rules where 000 -> 1 this can give issues, so you could make the edges wrap around. I just make the arraysize very large so the edge is less of an issue.

Annabelle Lee - January 17, 2017, 4:16 pm

(Jan 17, 2017 10:09 AM)Kantos Wrote: [CITE](#) Oh yeah, technically there isn't supposed to be an edge case since mathematically it's an infinite array. I just hold the two ends constant, i.e. I only change cells 1 through n-2 (0 and n-1 being the ends). For simulations that "grow" like Rule 90 generally you stop the simulation before it reaches the bounds so it's fine. For rules where 000 -> 1 this can give issues, so you could make the edges wrap around. I just make the arraysize very large so the edge is less of an issue. Ah, well I went for the wrap-around method, it makes the code a bit messier/slightly more complicated, but it does work fine. I implemented all 4 rules using 3 functions total. (main, sub-rules function and rule function.) Sub-rules is what is used for rules 34, 110, 184 in order to find the next 'cell' value in the next row. Rule function is what actually does the setting of the cell, and also houses the very simple XOR rule in it for #90 (why I don't have rule 90 in sub rules, it's not needed.) I abused ternary operator and you only need to change rule name in the function call at line #45 to change which is used. It is with C99 (setting int i during for loops is main reason.)

The whole thing clocks in at exactly fifty lines. I could make it a bit less if I remove the lines between the functions, but no matter.

Code #include <stdio.h> // implementation of cellular automata by Annabelle Lee of HF.net

```
#define MAX 33
#define rule30 1
#define rule90 2
#define rule110 3
#define rule184 4
```

```
int subrules(int l, int c, int r, int rule_number) {
    if (rule_number == rule30)
        return (l == 0 && r == 1 && c == 1) ? 1 : (l == 1 && r == 0 && c == 0) ? 1 : (l == 0 && r == 1 && c == 0) ? 1 : (l == 0 && r == 0 && c == 1) ? 1 : 0;
    else if (rule_number == rule110)
        return (l == 0 && r == 1 && c == 1) ? 1 : c ^ r;
```

```

else
    return (c == 0) ? l : r;
}

void rule(int *_pre, int *_new, int count, int rule_number) { // function which will take the rule chosen
above and use it.
    for (int i = 0; i < count; i++) {
        switch (rule_number) {
            case 1:
                _new[i] = (i == 0) ? subrules(_pre[i + 1], _pre[i], _pre[count - 1], rule30) : (i == (count - 1))
? subrules(_pre[0], _pre[i], _pre[i - 1], rule30) : subrules(_pre[i - 1], _pre[i], _pre[i + 1], rule30);
                break;
            case 2:
                _new[i] = (i == 0) ? _pre[i + 1] ^ _pre[count - 1] : (i == (count - 1)) ? _pre[0] ^ _pre[i - 1] :
_pre[i - 1] ^ _pre[i + 1];
                break;
            case 3:
                _new[i] = (i == 0) ? subrules(_pre[i + 1], _pre[i], _pre[count - 1], rule110) : (i == (count -
1)) ? subrules(_pre[0], _pre[i], _pre[i - 1], rule110) : subrules(_pre[i - 1], _pre[i], _pre[i + 1], rule110);
                break;
            case 4:
                _new[i] = (i == 0) ? subrules(_pre[i + 1], _pre[i], _pre[count - 1], rule184) : (i == (count -
1)) ? subrules(_pre[0], _pre[i], _pre[i - 1], rule184) : subrules(_pre[i - 1], _pre[i], _pre[i + 1], rule184);
                break;
        }
    }
}

```

```

int main() { // Main function.
    int previous_iteration[MAX] = { 0 }, new_iteration[MAX] = { 0 }; // An array which will hold the
previous (base iteration) // will wraps around
    previous_iteration[17] = 1;
    for (int i = 0; i < 16; i++) { // lets do fifteen generations being set.
        for (int j = 0; j < MAX; j++) {
            (previous_iteration[j]) ? printf(". ") : printf(" "); // we print the dots only, so it looks prettier
than 1 and 0.
        }
        printf("\n");
        rule(previous_iteration, new_iteration, MAX, rule90); /* CHANGE THIS RULE NUMBER
WHEN YOU WANT TO CHOOSE A DIFFERENT RULE! */
        memcpy(previous_iteration, new_iteration, sizeof(previous_iteration)); // set array from new to
old
        memset(new_iteration, 0, sizeof new_iteration); // zero out the array
    }
    return 0;
}

```

P.S. If you're compiling and running as an exe/clicking it instead of running from a terminal in linux or command prompt. Then you can insert a `getchar()` at the end of main.

Kantos - January 17, 2017, 7:22 pm

Interesting.

Surely your code could be simplified and made more general (include every rule) by doing what I

did though: for your subrule function, taking an unsigned value with the rule number and calculating what it should do from there rather than hardcoding them. Remember that the rules are named based on the binary value of their output representation.

E.g.

Code current pattern	111	110	101	100	011	010	001	000
new state for center cell	0	1	0	1	1	0	1	0

01011010 = 90

So for any rule, if I want to know what 001 produces, I just look at the second least significant bit. Say we take Rule 23, 23 = 00010111, so I know that the pattern 111 produces a 0, 000 produces a 1, 100 produces a 1, etc.

And yeah that is kinda ternary operator abuse lol, very unreadable. Don't feel constrained to 50 lines, I didn't do wraparound.

Annabelle Lee - January 17, 2017, 7:34 pm

(Jan 17, 2017 08:22 PM)Kantos Wrote: [\[CITE\]](#) Interesting.

Surely your code could be simplified and made more general (include every rule) by doing what I did though: for your subrule function, taking an unsigned value with the rule number and calculating what it should do from there rather than hardcoding them. Remember that the rules are named based on the binary value of their output representation.

E.g.

Code current pattern	111	110	101	100	011	010	001	000
new state for center cell	0	1	0	1	1	0	1	0

01011010 = 90

So for any rule, if I want to know what 001 produces, I just look at the second least significant bit. Say we take Rule 23, 23 = 00010111, so I know that the pattern 111 produces a 0, 000 produces a 1, 100 produces a 1, etc.

And yeah that is kinda ternary operator abuse lol, very unreadable. Don't feel constrained to 50 lines, I didn't do wraparound. Oh! That's a clever idea. I'm not sure why I didn't catch onto that. I'll work it into my code in the morning.

Kantos - January 17, 2017, 7:50 pm

I'm right on the brink of 50 lines anyway, and if I add wraparound or some easy optimizations it will no longer be under 50.

For instance, I have two char array buffers, an old state buffer and new state buffer. I read from the old(back) and write to the new(front), print the new, then copy the front to the back and repeat. I could probably optimize it slightly by "alternating" the buffers, i.e. read from the back, write to the front, print the front, then read from the front, write to the back, then print the back, repeat and I would save a memcpy() call.

Probably won't make a huge performance difference on large sizes but yeah I just wanted to mention that.

Annabelle Lee - January 18, 2017, 11:53 am

(Jan 17, 2017 08:50 PM)Kantos Wrote: [\[CITE\]](#) I'm right on the brink of 50 lines anyway, and if I add wraparound or some easy optimizations it will no longer be under 50.

For instance, I have two char array buffers, an old state buffer and new state buffer. I read from the old(back) and write to the new(front), print the new, then copy the front to the back and repeat. I could probably optimize it slightly by "alternating" the buffers, i.e. read from the back, write to the front, print the front, then read from the front, write to the back, then print the back, repeat and I would save a memcpy() call.

Probably won't make a huge performance difference on large sizes but yeah I just wanted to mention that.

I didn't worry about alternating the buffers, it seems too much effort for such a little payoff.

So this is my new code, with the magical power of bits and a lookup table.

Code #include <stdio.h> // implementation of cellular automata by Annabelle Lee of HF.net

```
#define MAX 33
#define RULENUM 90 // change this to depend on which rule you want
#define NTHBIT(x, n) x & (1 << n) // this will take the nth bit from the value of the number (x), easy
way to see the rules. (right to left, if it's set it's a number, else it's 0.)

int pattern_match(int l, int c, int r) { // looks for the left, right and centre numbers passed to it using a
lookup search in the array to find the nth bit to check for.
    int bp[35] = {l, c, r, 1, 1, 1, 7, 1, 1, 0, 6, 1, 0, 1, 5, 1, 0, 0, 4, 0, 1, 1, 3, 0, 1, 0, 2, 0, 0, 1, 1, 0, 0, 0,
0};
    for (int i = 3; i < 29; i++) // we'll go through the array starting at index i upto 31st
        if (bp[i] == bp[0])
            if (bp[i + 1] == bp[1])
                if (bp[i + 2] == bp[2])
                    return bp[i + 3];
}

void rule(int *_pre, int *_new, int count, unsigned int rule_number) {
    for (int i = 0; i < count; i++) {
        _new[i] = (NTHBIT(rule_number, pattern_match(_pre[i - 1], _pre[i], _pre[i + 1]))) ? 1 : 0;
    }
}

int main() { // Main function.
    int previous_iteration[MAX] = { 0 }, new_iteration[MAX] = { 0 }; // An array which will hold the
previous (base iteration) // will wraps around
    previous_iteration[17] = 1;
    for (int i = 0; i < 16; i++) { // lets do fifteen generations being set.
        for (int j = 0; j < MAX; j++) {
            (previous_iteration[j]) ? printf(". ") : printf(" "); // we print the dots only, so it looks prettier
than 1 and 0.
        }
        printf("\n");
        rule(previous_iteration, new_iteration, MAX, RULENUM); /* CHANGE THIS RULE NUMBER
WHEN YOU WANT TO CHOOSE A DIFFERENT RULE! */
        memcpy(previous_iteration, new_iteration, sizeof(previous_iteration)); // set array from new to
old
        memset(new_iteration, 0, sizeof new_iteration); // zero out the array
    }
    return 0;
}
```

```
}
```

A bit about it: First, I don't abuse the ternary as much, simply using it in two locations to save some space. The function pattern match has a lookup table in which the left, center and right values are saved, and then compared with the rest of the table. Using this, the table then gives us the nth bit that corresponds to that pattern in an unsigned integer. Returning that. That is then passed to the rule function which simply used that NTHBIT macro to see if it is zero or none zero, and then sets the new cell in response to that.

It could possibly be shortened/optimized a bit further, but it wouldn't gain much from it. Currently the code clocks in at a mere 35 lines. (I just couldn't resist that 50 line limit, too much fun.) A change from the first version is that I disabled the wrap around, since the generations are stopped before it ever reaches that point.

Kantos - January 18, 2017, 3:03 pm

I'm slightly confused on how that pattern match function works, what are those values in the bp array?

Annabelle Lee - January 18, 2017, 3:11 pm

(Jan 18, 2017 04:03 PM)Kantos Wrote: [CITE](#) I'm slightly confused on how that pattern match function works, what are those values in the bp array?

Code int pattern_match(int l, int c, int r) { // looks for the left, right and centre numbers passed to it using a lookup search in the array to find the nth bit to check for.

```
int bp[35] = {l, c, r, 1, 1, 1, 7, 1, 1, 0, 6, 1, 0, 1, 5, 1, 0, 0, 4, 0, 1, 1, 3, 0, 1, 0, 2, 0, 0, 1, 1, 0, 0, 0, 0};
```

```
for (int i = 3; i < 29; i++) // we'll go through the array starting at index i upto 31st
```

```
if (bp[i] == bp[0])
```

```
if (bp[i + 1] == bp[1])
```

```
if (bp[i + 2] == bp[2])
```

```
return bp[i + 3];
```

```
}
```

Okay so array bp is basically a lookup table of patterns and the nth bit. The first three numbers, l, c and r are basically passed to the function and stored there for easy comparison. Tho looking at it I could've just skipped on that entirely and kept them as is. Regardless the table is formatted as follows:

The 1 1 1 7 is the pattern you're searching/comparing for. And 7 is the nth bit. That is when you have unsigned int of say 90.

What happens is you get 01011010. From left to right the bits are ordered as 7 through 0. Once you get that bit number, you return it. Then you simply use the NTHBIT macro to get the 7th bit in the number 90 is zero, so you return that. If the NTHBIT returns a none zero value, you know that the NTH bit is 1.

Kantos - January 18, 2017, 3:32 pm

Ahhh I see now, interesting.

Here is my code by the way:

I generally use very large sizes and write the result to file, then zoom way out on a GUI editor. I want to eventually extend this to write monochrome image files instead so I can do other cellular automata within one program but this was just a quick project.

