1.1 Design on the Fly: Experiments with Complex Systems – by Tony Smith

1.1.1 Abstract

Complex systems research was given a new impetus by Wolfram's 2002 publication of *A New Kind of Science.* NKS refocused attention on cellular automata (CA) and other complex systems in which a "simple program" is applied in parallel to determine the next state of each of many locations based on the previous states of neighbouring locations. While Wolfram added special CA functions to his Mathematica™ software, I chose to continue independent research using my "Swiss Army knife"—Perl. Even simple experiments can produce vast amounts of data for subsequent analysis and until you see what you've got you may only have a vague idea as to what you might want to do next, so there is no room for a traditional design and development cycle. It is very easy to produce so much data that the capacity of the system is quickly exceeded, requiring the invention of efficiencies before research can proceed further. Data originally managed in plain text files has progressively migrated to database and object representations. Perl encourages flexible analysis and presentation of experimental data. While much of my complex systems research has focused on CA, this paper focuses on my use of Perl to study a particular evolving network which I call "Tick Tock".

1.1.2 Cellular Automata

In 1983 while reviewing an obscure "transportable" computer equipped with GW Basic I came across a description of Ed Fredkin's simple self replicating cellular automata (CA) in the Mathematical Games column of *Scientific American* and felt implementing it would make a good test. To that point I had not been aware of the deep significance of CA, a field initially popularised by John Conway's *Game of Life.*[1] However, within 3 years my generalisation of Ed Fredkin's CA, *Pattern Breeder*, had been featured in the successor Computer Recreations column.[2]

CAs and comparable complex systems evolve over any number of generations by the repeated application of some or other rule based on the state of a local neighbourhood which is used to determine the next local state, typically in parallel, for every location in some "universe". CAs are distinguished by having their local state values arranged on a regular grid, most often a two dimensional rectangle. Frequently the grid cells are in one of two colours, black or white, representing on or off, live or dead.

The CA rule for Conway's *Life* readily produces quite a range of simple stable, oscillating or moving *Life forms* so the hunt was soon joined to invent a sufficiently complex multicoloured CA rule in which a self replicating pattern could be engineered. Fredkin's rule trumped that effort by showing that a very simple two colour rule could also achieve self replication. The rule just needed to set a cell on if there was an odd number of live cells in its neighbourhood and off if there was an even number. It could be easily implemented in those days of early black and white displays by successively XORing displaced copies of the screen buffer.

Despite its aesthetically pleasing results, *Pattern Breeder* did not provide any sustained intellectual challenge. So from 1987 to 2003 almost all my bursts of hard research into complex systems were in marginally constrained *Life* universes, using more and more of my own Perl code in more recent times. Two ongoing projects are introduced in Appendices B and C.

1.1.3 Complex Systems Theory

Also in 1983 Stephen Wolfram found that the 256 possible three neighbour, two colour, one dimensional CA Rules could all be categorised into what are still well known in the field as Classes 1 through 4,[3] definitions he restated in 2002:[4]

- In class 1, the behavior is very simple, and almost all initial conditions lead to exactly the same uniform final state.

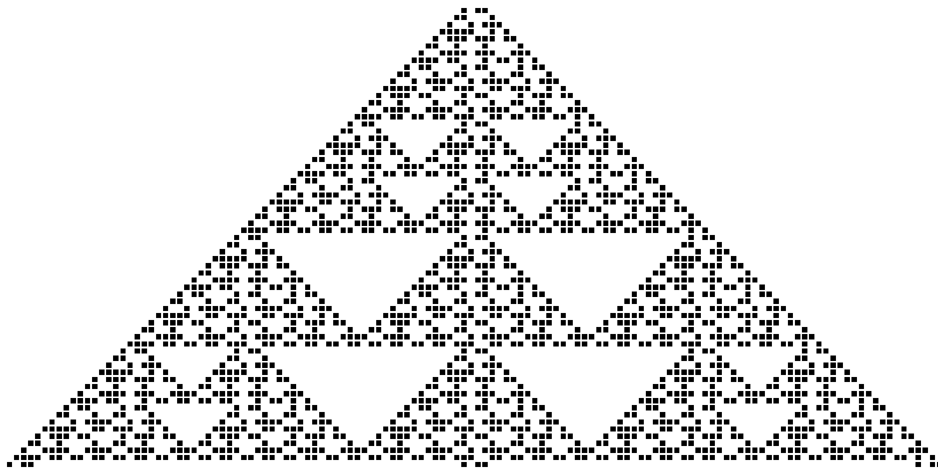


Fig. 1: Wolfram's Rule 150 provides a one dimensional analogue of the Fredkin CA, starting with a simple seed pattern at the top and showing 64 more generations down the page. The Wolfram rules are based on a neighbourhood of 3 cells. Rule 150 sets a cell on if there are 3 or 1 live cells in its neighbourhood at the previous tick. The seed pattern is clearly replicated every 8th generation.

- In class 2, there are many different possible final states, but all of them consist just of a certain set of simple structures that either remain the same forever or repeat every few steps.
- In class 3, the behavior is more complicated, and seems in many respects random, although triangles and other small-scale structures are essentially always at some level seen.
- (...) class 4 involves a mixture of order and randomness: localized structures are produced which on their own are fairly simple, but these structures move around and interact with each other in very complicated ways.

Wolfram quickly established himself as a leader in the Complex Systems[5] movement but by 1986 he had also started work on Mathematica™.[6] This provided reason for our paths to cross at times during the ensuing three years before we each headed off in our own directions for another decade.

As well as developing all the obviously necessary functionality, Wolfram enhanced Mathematica to serve as his Swiss Army knife for tackling the vast scope of experiments with simple programs and more that all went into the production of his epic work *A New Kind of Science* (NKS).[7] The 2002 publication of NKS may be finally helping the study of CA and similar to cast off the "recreation" tag.

One and two dimensional CA fit human visual perception better than other well studied complex systems, in part because they presume conventional notions of space and time. Those of us who entertain the hope that experiments with simple programs will shed light on deeper questions would like space and time to emerge from something simpler. A popular candidate for that simpler something is a Planck-scale[8] evolving network in which the bottom level is defined by elementary nodes and links. If they are truly elementary as suggested, it makes no sense to ask what those nodes and links are made of. This paper looks at the emulation of a simple evolving network I call "Tick Tock".

1.1.4 Simple Graphs and Simplexes

Mathematicians call such elementary networks of nodes and links "simple graphs". Their study is a part of "graph theory" which has nothing to do with what most people think of when "graphs" are mentioned. In the language of graph theory we also talk about "vertices" and "edges" in lieu of "nodes" and "links", by natural analogy with solid geometry, another area that has had more than it's share of my time.

The class of graphs which are maximally connected for a given number of nodes are known as

"simplexes". The triangle is the first non-trivial simplex, technically a 2-simplex because it can be represented in 2 dimensions without edges crossing. The tetrahedron is thus a 3-simplex and the 4-simplex is known as a "pentatope",[9] but beyond that they just get their numbers. Simplexes have proven to be of special importance in understanding *Tick Tock.*

The initial method I chose to generate starting graphs was to generate the complete simplex for **$nodes** and then randomly chose the desired **$edges** from the pool of **$nodes * ($nodes - 1) / 2** edges of the simplex. While grappling with the potential significance of that tiny fraction of early runs which "ran away", I thought I might try a more systematic survey of initial seed graphs than was likely to be achieved by random selection. Though I already knew that enumerating all topologically distinct simple graphs is a non trivial but well studied problem, I still went off and tried to build my own tables. That exercise turned out to have one lasting benefit of getting me to finally keep some of my results in database tables.

In the end it was mostly complete simplex starting configurations which provided real insight into the surprising behaviour of *Tick Tock.*

1.1.5 Doing it in Perl

As an old school analyst-programmer, I quickly became interested in doing things on the Web than needed CGI and found myself maintaining and soon developing back-end systems in Perl[10] from where I have slowly absorbed much of the broader culture in which Perl is situated.[11] This paper is about using Perl to study one particularly simple evolving network, *Tick Tock*, an investigation I found I could no longer postpone early in 2004.

This paper is not about *Tick Tock* per se. More than you might ever want to know is already on the *Tick Tock* website,[12] though that line of research is far from finished. Rather it is about why Perl's "do what I mean" philosophy and low coding overheads are well suited to this kind of research.

My main development computer is an aging 400 MHz G4 PowerMac running the current version 10.3.6 of OS X.[13] Most of my editing is done in BBEdit[14] 6.1.2 which is three years out of date but which still plays nicely with Perl. The G4 is running MySQL 4.0.18 and has installed a limited set of CPAN and private modules, the latter mostly still evolving or client-specific.

The only private module that is currently useful to my complex systems research is **GIF::Generate** [15] to which I add features as needed and which was used to produce images for the *Tick Tock* website from the kind of results data that this paper is mostly concerned with.

1.1.6 In Search of an Hypothesis

Many areas of research benefit from the methodological discipline required to propose an hypothesis and design experiments to test that hypothesis. This is not one of those areas. At this still early stage in the study of this class of simple programs–complex systems my methodology remains:

- chose a simple enough rule
- implement
- see what happens
- draw unexpected conclusions

So far this seems to work, but the point of this paper is not the method nor the conclusions, but rather why Perl has proved such a productive way to tackle some potentially tricky implementations.

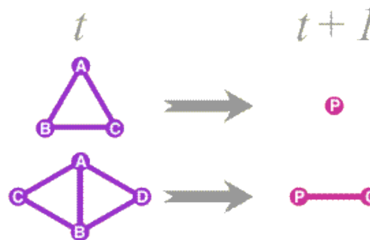
1.1.7 The *Tick Tock* Rule Defined

Tick Tock is a simple program acting on the nodes and edges of a graph to produce a new graph each "tick", the evolving series of graphs over a number of ticks behaving as a Class 2 complex system, at least for sufficiently connected starting graphs. (Graphs which are more sparsely connected behave as Class 1, quickly dying or stabilising.)

The local *Tick Tock* mechanism can be defined completely in two English sentences:

- Every triangle in the graph at one tick is succeeded by a node at the next tick.
- Those pairs of nodes which succeeded two triangles which shared an edge at one tick are linked by a new edge at the next tick.

Or, in a simple enough graph transformation diagram:



Or, in four formal lines of mathematical symbology, as also shown on the front page of the *Tick Tock* website.[see 11]

1.1.8 Implementing a "Naive" Algorithm

A naive algorithm for *Tick Tock* based directly on the above definition can be represented in Perlish pseudo code where:

```
$a, $b, ... are nodes at t; $p, $q, ... are nodes at t+1
@edges is a list of node pairs representing the edges at t and at t+1
%dests is a hash by node of lists of the other ends of the edges at t
@faces is a list of node triples in which each pair is an edge at t
%oppos is a hash by edge of the opposite nodes of each face containing that edge
%child is a hash by triple at t of generated nodes at t+1
```

With just **@edges** preinitialised, each successive tick is computed by:

```
foreach $a:$b in @edges {push %dests[$a], $b; push %dests[$b], $a}
foreach $a:$b in @edges {
    foreach $c in %dests[$a] and %dests[$b] {
        push @faces, $a:$b:$c
        push %oppos[$a:$b], $c for each 3 pairs from $a:$b:$c
    }
}
foreach sort @faces {push %child[$a:$b:$c], $p}
foreach $a:$b:$c => $p in %child {
    foreach $a:$b:$d in %oppos where $d&ne;$c {
        $q = %oppos[$a:$b:$d]
        push new @edges, $p:$q
    } for each 3 pairs from $a:$b:$c
}
```

Even this core tick iteration takes around 50 lines of actual code Perl to fully implement, plus more lines to do initial setup, to test for termination conditions and to report results. The **ticktock.pl** listing shown in Appendix A is a relatively early, relatively stable version of Perl code which implements this naive algorithm.

1.1.9 Some Early Results

Repeatedly running **ticktock.pl** with **$nodes = 5** and **$edges = 7** produces a random sequence of results of the form:

```
[TSG4:~/Data/TickTock] ynotds% perl ticktock.pl
0 (5 nodes, 7 edges):
1 (2, 1):
  node edges: 1 x 2
  edge faces: 1 x 4, 2 x 1
Took 0 seconds!
[TSG4:~/Data/TickTock] ynotds% perl ticktock.pl
0 (5 nodes, 7 edges):
1 (3, 2):
  node edges: 1 x 2, 2 x 1
  edge faces: 1 x 5, 2 x 2
Took 0 seconds!
[TSG4:~/Data/TickTock] ynotds% perl ticktock.pl
0 (5 nodes, 7 edges):
1 (3, 3):
  node edges: 2 x 3
  edge faces: 1 x 6, 3 x 1
2 (1, 0):
  node edges:
  edge faces: 1 x 3
Took 0 seconds!
[TSG4:~/Data/TickTock] ynotds% perl ticktock.pl
0 (5 nodes, 7 edges):
1 (4, 6):
  node edges: 3 x 4
  edge faces: 2 x 6
2 (4, 6):
  node edges: 3 x 4
  edge faces: 2 x 6
Took 0 seconds!
```

Most of these indicate that a starting network with 5 nodes and 7 links dies almost instantly under the *Tick Tock* rule:

- The first randomly selected set of 7 links must have formed only two triangles with just one shared edge and thus generated a pair of nodes with a single link at t+1, just as illustrated by the graphical representation of the rule (above).
- The second formed three triangles, one sharing edges with each of the other two, from which it generated three nodes, two end nodes linked to a central node.
- The third initially formed three triangles at t=0, which in turn formed one triangle at t+1, this forming a single unlinked node at t+2.
- It is the fourth and final result shown which is just a little more interesting, having terminated because it had detected the persistence of a network with 4 nodes each belonging to 3 edges and 6 edges each belonging to 2 faces—a tetrahedron!

Of course I was already aware through my familiarity with solid geometry that a tetrahedron would generate a successor tetrahedron under the *Tick Tock* rule. I needed to push on beyond 5 nodes and 7 links to start finding surprises.

Historically this code was rarely run more than a few times without needing some or other tweak, and it was originally mostly run from BBEdit's Perl menu anyway, so **$nodes** and **$edges** were easily changed in the code between runs. The basic procedure was to set **$nodes** and **$edges**, run a few times and try to see something in the output before trying new values or, as often as not, tweaking the loop termination test or the reporting procedure.

1.1.10 Dealing with the Expected Unexpected

It is easier to demonstrate than to record any more of the story of the gradual production of evidence because as motivating as it was to do it the first time, it would bore anybody to distraction

to repeat, especially to repeat the false starts and detours, most of which are omitted from this account.

Other than the obvious fact that relatively sparse graphs would die out, I had no clear expectations beyond the persistence of isolated tetrahedra. So when a smattering of seeds with **$nodes > 5** "ran away" it took me a while to work out what was going on. My first tactic was to add new loop terminating conditions and then to save the prematurely terminated graph data to a file for manual examination, data that usually looked a lot worse than:

```
0:1:2 0:1:3 0:2:3 1:2:3 2:3:4 2:3:5 2:4:5 3:4:5 4:5:6 4:5:7 4:6:7 5:6:7 6:7:8 6:7:9 6:8:9
7:8:9 8:9:10 8:9:11 8:10:11 9:10:11 10:11:12 10:11:13 10:12:13 11:12:13 12:13:14 12:13:15
12:14:15 13:14:15 14:15:16 14:15:17 14:16:17 15:16:17 16:17:18 16:17:19 16:18:19 17:18:19
```

That ideal data is generated by just one of the 1365 possible cases where **$nodes = 6** and **$edges = 11**, the one seeded with a set of links which define two tetrahedra **0:1:2:3** and **2:3:4:5** sharing one common edge. In this rare case, the 20 nodes at t+3 form 36 triangles and thus generate 36 nodes at t+4. Working from random seeds it was a slow and painful process to work out that most early runaway configurations contained what we might describe geometrically as long lines of edge connected tetrahedra. I now know how to explain these directly from the *Tick Tock* rule,[16] but that understanding was a while coming.

1.1.11 Spotting Patterns in Early Results

I generated, reprocessed and stored away a large quantity of data before I started to comprehend some larger patterns. Some of this required reading old results back in from text files, parsing them, doing further processing and generating yet more text files. However a lot of the results were to eventually be presented via John N Huffman's ancient ChemRote applet[17] which uses plain text data files to specify the 3-D position of nodes and links, e.g. ChemRote data defining a regular tetrahedron is:

```
CARTESIAN   4   6   MSC00000
vx(1)   1.000   1.000   1.000   1
vx(2)   1.000  -1.000  -1.000   1
vx(3)  -1.000   1.000  -1.000   1
vx(4)  -1.000  -1.000   1.000   1
ENDATOMS
  1    2
  1    3
  1    4
  2    3
  2    4
  3    4
ENDBONDS
```

Conceptually, *Tick Tock* nodes and links do not have spatial coordinates but it does make it a lot easier to show them to humans if you pretend that they do. Over time I merged the generating logic of **ticktock.pl** with the generation and parsing of Chemrote-style coordinates and generalised from there to a comparable process using n-dimensional coordinates.[18] Then I wrote scripts to compute frequency distributions of the 3-D and n-D lengths of edges, often just as a sanity check.

Eventually I came to realise that the *Tick Tock* rule can only really produce a network of edge-joined simplexes.[19] I had already produced standalone scripts to detect tetrahedra and similar amongst the triangular face data which had become the default way of representing the shape of the graph after each tick. Another hard-won discovery was that despite the network graph being fully generated each tick, even within rapidly inflating networks ever larger patterns persisted.

1.1.12 Starting Points for Refactoring

I was about ready to have a first cut at consigning that shape data to database tables. The trick was going to be to build tables identifying not just triangular faces but also edges, tetrahedra, higher simplexes and notional n-D node coordinates. A partially recursive second generation algorithm[20] could then take advantage of the emergent properties of tetrahedral persistence and edge-to-simplex generation to track the evolution of what I was already seeing to be quite complex structure, especially beyond the 4th tick from a 6-simplex seed, where the amount of raw data needed to record the network configuration continues to grow exponentially:

```
    2850  6-simplex_t1_raw
   29403  6-simplex_t2_raw
  198351  6-simplex_t3_raw
 1004427  6-simplex_t4_raw
 4583968  6-simplex_t5_raw
20819989  6-simplex_t6_raw
```

As long as the data stayed within bounds dictated by physical memory, deeper exploration only required patience as each new tick of the 6-simplex took more than 4 times as long as the previous tick. The longest runs remained measured in minutes until the limits of physical memory were exceeded. But almost as soon as any of the by then very large arrays and hashes used to keep track of the evolving network spilt into virtual memory things started to get ugly.

1.1.13 Virtual Memory Hits Some Limits

For most things we use computers for, the algorithms which manage virtual memory can be surprisingly efficient, to the point where you usually don't notice them. Even when you do it is just because selecting some function you haven't used for a while may take a couple of seconds to page everything it needs back into physical memory, especially when you have a lot of tasks running.

Interestingly, *Tick Tock* and comparable CA experiments seem to gain nothing from such virtual memory smarts, to the point where their performance could only be described as pathological, slowing to something like one tenth of their previous speed almost as soon as virtual memory comes into play. This is likely to be because the application needs to have a large array or hash entirely in memory to update it or whatever, but ultimately for this class of problem it would not matter how smart you made the interpreter.

The nature of *Tick Tock* et al is that the next data point generally cannot be predicted by any simpler process. In the odd case where you discover a computational short cut, you tend to apply it within the generation process, maybe gaining a useful increment in the parameter space that you can get results for. But before long even such tuned processes will hit the same wall in any finite machine, so the qualitative difference between the results I can squeeze out of my aging G4 and the results I might get my throwing the same problem at e.g. Virginia Tech's cluster[21] of 1100 XServes may not always be significant.

1.1.14 Pursuing Multiple Objectives in Parallel

During this kind of research different parts often seem to run in overlap. You keep running old processes to fill in gaps in your data set while working on new approaches to generating more data, on better representations of the data you have at hand, and on finding deeper patterns in the data.

It took a while to settle down the first useful cut of a process to generate database tables. Both the database schema and that first cut process were only properly designed to work with pure simplex seeds upto and including but not beyond the 6-simplex. At the time of heading in that direction I had still not fully appreciated the rich results coming from the 6-simplex seed nor the potential importance of higher simplex seeds, except that each step higher on that dimension would greatly increase the processing load.

The most testing moment came when I asked **db_ticker.pl** to generate the 7th tick data. Extrapolating from the previous ticks I had estimated it might run for up to 48 hours and so launched it when I was leaving for a weekend away from the computer, first taking note of the state of the few indicators I could find of task progress. When I got back I soon realised that while it had

got a lot done, it was not about to finish any time soon, and I was able to switch to doing all my non-long-run work on my iBook, even using a window there to monitor progress on the G4 and sometimes grabbing data snapshots.

This was clearly not the time for further developing **db_ticker.pl** nor for launching any other potentially long running processes. However it was also a blessing in disguise as it forced me to start trying to get the results in hand into some kind of order, as well as to keep thinking about the overall patterns that had emerged. I did manage to keep updating my estimated finishing time for the run that was still in progress, but still was not all that confident by the next weekend. However it did finally finish after around ten days and left me with some database tables that were big enough to be a bit of a worry in themselves:

```
  81484900  nodes.MYD
  21996544  nodes.MYI
  98659152  edges.MYD
 122805248  edges.MYI
 119076237  triangles.MYD
 155422720  triangles.MYI
  48024700  tetrahedra.MYD
  60545024  tetrahedra.MYI
   7167475  pentatopes.MYD
  10753024  pentatopes.MYI
```

Before I started that long final run, I was already a fair way down the track in quantifying persistent symmetries within the network inflating from a 6-simplex seed. Well before the long run had finished I was well aware that it really wasn't going to tell me anything that could not be extrapolated from running the process on basically 1/210th of the network, the whole network having evolved quickly to form 210 identical "joining units" connected by 140 boundary tetrahedra. All the growth from tick to tick happens, and some significant asymmetry emerges, inside the joining units.

1.1.15 Unfinished business

For what started out intended to be just a minimalist experiment to help me and possibly others start to better understand evolving networks, *Tick Tock* has already delivered more than its share of surprises. I eventually reached a point where I felt I could tell enough of the story with confidence that the loose ends could wait a while. So I generated many ChemRote datasets, a few density plot GIFs and a lot of words to produce a website that I felt was good enough to be worth showing to others who might be interested in the outcomes rather than the process.[see 12]

The story of the journey I have shared here is not so much a story about the theory behind *Tick Tock* as it is an ode to Perl as an environment in which I could start from scratch on a project of this magnitude and produce very useful results from a couple of months of part time endeavour. Perl is clearly a great way to go when you don't know where you're going nor where the finish line might be.

I now understand the emergent phenomena of *Tick Tock* well enough that I can have confidence in some formulae I have developed to predict the number and size of joining units formed by higher simplexes seeds. However I do need to refactor the database schema and **db_ticker.pl** to deal both with those higher simplexes and with the evolution of joining units in isolation. Only after that can I be sure whether there are any more surprises and more extensively survey the emergence of asymmetry within higher simplex joining units, where I feel I am currently a bit too dependent on the singular 6-simplex data set.

For *Tick Tock*, for the CA experiments introduced in Appendices B and C and for future experiments, one thing I am trying to conceptualise is a high level scheduling process. Such a process would allow queuing of a number of sets of experimental parameters for which results are sought, each involving run times measured in anything from minutes to days. Without something like that it is a practical impossibility for an experimenter to keep this type of data generation inching forward, at least not short of continuing to give the individual runs a level of attention that can only really be

justified during early exploration of new problem space.

1.1.16 Appendix A: ticktock.pl

```
#!/usr/bin/perl
# Explore Tick Tock Rule
# Copyright 2004, Tony Smith

use strict;
my $t0 = time;

# set up initial working set of edges

my $nodes = 5;
my $edges = 7;
print "0 ($nodes nodes, $edges edges):\n";

my @possible; # edge pool
for (my $i=0; $i<$nodes; $i++) {
    for (my $j=0; $j<$i; $j++) {
        push(@possible, "$j:$i");
    }
}
my @pairs; # select edges,
my @graph; # list connected nodes by node
while (@pairs < $edges) {
    my $pick = int(@possible * rand());
    my ($picked) = splice(@possible, $pick, 1);
    push(@pairs, $picked); # select edge
    my ($p, $q) = split(/:/, $picked);
    push(@{$graph[$p]}, $q); # list connected nodes by node
    push(@{$graph[$q]}, $p);
}
my @triples; # detect triangular faces,
my %edges; # list third triangle vertices by edge,
my %index; # give new node number to face

# determine new nodes (triangular faces) and edges (adjacent faces)

foreach my $tick (1 .. 10) {
    (@triples, %edges, %index) = ();
    my @prev = ($nodes, $edges);
    foreach my $edge (@pairs) { # detect triangular faces
        my ($p, $q) = split(/:/, $edge);
        my $max = $p > $q ? $p : $q;
        my @plist = my @qlist = ();
        foreach (@{$graph[$p]}) {push(@plist, $_) if $_ > $max}
        foreach (@{$graph[$q]}) {push(@qlist, $_) if $_ > $max}
        for (my $i=0; $i<@plist; $i++) {
            for (my $j=0; $j<@qlist; $j++) {
                if ($plist[$i] == $qlist[$j]) { # detect third triangular vertex
                    my $r = $plist[$i];
                    push(@triples, "$p:$q:$r"); # triangular face
                    push(@{$edges{"$p:$q"}}, $r); # list third triangle vertices by edge
                    push(@{$edges{"$p:$r"}}, $q);
                    push(@{$edges{"$q:$r"}}, $p);
                }
            }
        }
    }
    $nodes = my @next = sort { # ordering makes it easier to spot persistent patterns
        my @a = split(/:/, $a);
        my @b = split(/:/, $b);
        $a[0] == $b[0] ? $a[1] == $b[1] ?
            $a[2] <=> $b[2] : $a[1] <=> $b[1] : $a[0] <=> $b[0];
    } @triples;
    last unless $nodes;
    for (my $i=0; $i<$nodes; $i++) {$index{$next[$i]} = $i} # give new node number to face
    @pairs = (); # new edges
```

```
    @graph = (); # list connected nodes by node
    for (my $i=0; $i<$nodes; $i++) {
        my $picked = $next[$i];
        my ($p, $q, $r) = split(/:/, $picked);
        pair($p, $q, $r, $i); # detect adjacent faces (new edges)
        pair($p, $r, $q, $i);
        pair($q, $r, $p, $i);
    }
    $edges = @pairs;

# report some basic statistics

    print "$tick ($nodes, $edges):\n";
    my (@counts, %freq) = (); # node edge count frequency
    for (my $i=0; $i<$nodes; $i++) {$freq{@{$graph[$i]} + 0}++ if ref $graph[$i]}
    foreach (sort {$a <=> $b} keys %freq) {push (@counts, "$_ x $freq{$_}")}
    print '  node edges: ', join (', ', @counts), "\n";
    @counts = %freq = (); # edge triangle count frequency
    foreach (keys %edges) {my $count = @{$edges{$_}}; $freq{$count}++}
    foreach (sort {$a <=> $b} keys %freq) {push (@counts, "$_ x $freq{$_}")}
    print '  edge faces: ', join (', ', @counts), "\n";

# detect stability or runaway growth

    if ($prev[0] == $nodes and $prev[1] == $edges or $nodes > 1000) {
        if ($ARGV[0]) {
            open OUT, '>', $ARGV[0]; # raw triangles for analysis
            print OUT join (' ', @next), "\n";
            close OUT;
        }
        last;
    }
}

print 'Took ', time-$t0, " seconds!\n";
# subroutines

sub pair { # detect adjacent faces (new edges)
    my ($p, $q, $r, $i) = @_;
    foreach (@{$edges{"$p:$q"}}) {
        unless ($_ == $r) { # ignore current face
            my $triple = join(':', sort {$a <=> $b} ($p, $q, $_)); # adjoining face
            my $j = $index{$triple}; # new node number for face
            push(@pairs, join(':', $i, $j)) if $i < $j; # new edge
            push(@{$graph[$i]}, $j); # list connected nodes by node
        }
    }
}
```

1.1.17 Appendix B: *Life in a Tube*

Simple seed patterns in an unbounded *Life*[see 1] universe almost always eventually stabilise to a collection of familiar "still lifes", oscillators and escaping "gliders". Random seeding of a larger rectangular *Life* universe also almost always stabilises to a similar collection of simple "life forms".

Others have independently discovered hints of the vast range of interesting results that emerge from *Life* confined to a narrow cylindrical universe. Many simple seed patterns never stabilise completely in the universe of *Life in a Tube.* Most of those which do not stabilise are kept going by the fact that a complete "leading edge" circumference of live cells around a tube, immediately adjacent to an area of empty cells, creates a new complete circumference of live cells in the adjacent circumference at the next tick. In the jargon of CA, the circumference moves at c. Seeding *Life in a Tube* with such a leading edge plus a simple pattern to cause interference behind that edge typically results in a trail of the products of that interference following after the edge which is advancing down the tube at c.

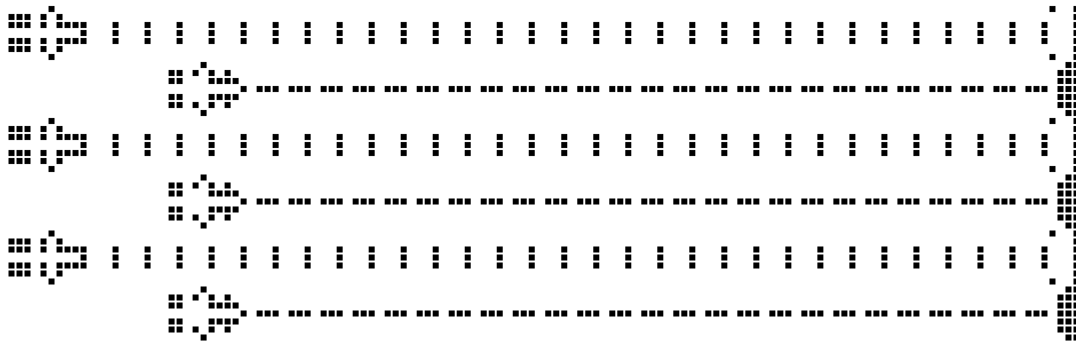


Fig. 2: The trail left after 302 generations of Conway's Life in a Tube of circumference 14 cells from a simple seed—one circumference of 14 live cells with an immediately adjacent half-circumference of 7 live cells. The Tube is shown rolled out 3 times to assist human perception. Each circumference leaves an initial double trail of "blinkers"—Life's common period 2 oscillators. "Blinker gobblers" which mop up the blinker trails at 2/3 c are formed during stabilisation, first for one trail and then for the other, after a sufficiently long interval to avoid mutual interference. This is one of the simpler of many interesting results generated from simple seeds by Life in a Tube.

Such trails of "debris" can be produced by an often long period cycle behind the leading edge moving at c or by a randomising process moving at average less than c and sustained by an ever-widening active zone behind the leading edge.[22] I still need to produce a more comprehensive presentation of *Life in a Tube,* but the purpose of introducing it here is that it was my first complex systems research where Perl played a role and it was the stepping off point for the stronger example discussed below in Appendix C.

I had first found evidence for some of the riches of what I now call *Life in a Tube* in the late 1980s, but it was not until more recent releases of Andrew Trevorrow's LifeLab,[23] an excellent implementation of *Life* for the Macintosh, that I started to systematically examine the evolution of patterns from a large range of seeds. By that time I was comfortable enough with Perl that I also implemented my own *Life* engine specially optimised so as to compute only the moving active zone of *Life in a Tube.*

Ultimately the primary way to present the results of *Life in a Tube* has to be animated two colour GIFs, so I developed **GIF::Generate** to handle as much of the GIF specification as I needed for that and some other purposes.[see 15] It turned out that my Perl code could not compete with LifeLab for the sometimes thousands of generations that needed to be run before the start of a sequence that I wanted to keep in an animation. Fortunately LifeLab saves generated patterns in RLE format which is easy to parse in Perl.

So I finished up combining my RLE parser, a revised version of my *Life* engine and my GIF generator to produce a CGI script which animates selected generations of *Life in a Tube.*[24]

While studying one of the simplest cases with a random component, evidenced by it slowly producing one or other of two possible patterns each 16 cells along the tube in random sequence, I found another use for Perl to directly code a much quicker but equivalent way of determining that particular random sequence. Until now it has been easier in the general case for a human working with LifeLab to classify the outcomes of a series of seed patterns, but that should be revisited at some stage.

1.1.18 Appendix C: *Trapper*

Runs of *Life in a Tube* in universes with diameter up to ten cells frequently finish up forming blocking patterns which seemingly repel the kind of advance of a complete circumference of live cells that powers *Life in a Tube.* Sufficiently often to not escape notice two or more such

"blockades" form in such a way that they enclose a stretch of tube which only contains whole circumferences which are either live or dead. And within those "traps" the configuration often stabilises into cycles, the periods of which would be considered unlikely in other *Life* universes.

It has been long known in CA circles that evolution from a single circumference of a toroidal *Life* universe follows Wolfram's 1D Rule 22 and the evolution of every second cell viewed at every second generation of Rule 22 follows Rule 90. So *Trapper* follows Rule 22 save and except for special end conditions immediately adjacent to the "blockades". If the blockade was treated as a third colour, a number of three colour 1D rules are seen to be exactly equivalent to *Trapper*.

This all added up to a prospect of developing a process for surveying the evolution of various seed pattern configurations in various width traps that would be much more efficient than examining them by hand in LifeLab or anything similar could be. At any given trap size, each possible seed pattern has a unique successor at t+1, so the first task was to start thinking in binary and mapping these successor relationships:

```
0=>0 1=>1
0=>0 1=>3 2=>3 3=>0
0=>0 1=>3 2=>7 3=>4 4=>6 5=>5 6=>1 7=>0
0=>0 1=>3 2=>7 3=>4 4=>14 5=>13 6=>9 7=>8 8=>12 9=>15 10=>11 11=>8 12=>2 13=>1 14=>1 15=>0
```

Of course the number of possible patterns inside a trap doubles for every increase of one in trap width, so even drawing diagrams by hand based on the complete set of successor relationships for a trap becomes problematic for widths greater than eight. But even in the width 4 data shown in the last line above, the broad principle of evolution towards persistent loops starts to show through:

```
5=>13=>1(=>3=>4=>14=>1)
10=>11=>8(=>12=>2=>7=>8)
6=>9=>15(=>0)
```

Perl also made it extremely easy to collect and present statistics on the size and number of persistent loops and the "hair" distance of other patterns from their eventual loops, including the nominal all dead loop:

```
4:  Loops: 4x2
    Hair:  2 2
    Dies: 1 1 1 1
5:  Loops: 4 1
    Hair:  2
    Dies: 1 2 4 5 3 2 4 4
6:  Loops:
    Hair:
    Dies: 1 3 3 5 5 7 14 10 8 2 2 2 2
7:  Loops: 12 4 1
    Hair:  18 12 10 10 6 8 4 2 4
    Dies: 1 4 4 1 1 3 5 6 4 2 2 2 2
8:  Loops: 12x2 4x3
    Hair:  55 30 30 32 16 18 12 8 4
    Dies: 1 6 8
9:  Loops: 12 8 4x4 1
    Hair:  62 56 65 66 34 32 28 8 6 10 6 4 6 6 12 10 4 4 10 8 4 6 6
    Dies: 1 9 9 2 1
10: Loops: 8x2 5 4x3
    Hair:  56 72 113 99 73 74 74 54 48 62 42 24 16 16 22 18 20 14 28 10 4 2 6 2 2
    Dies: 1 13 12 9 5
```

The index of the frequency counts for "Hair" and "Dies" corresponds to the number of successor patterns on the track to stability.

The predominance of loops with a period that is a multiple of 4 is due to strong "pressure" for the

trap to follow Rule 90, a pressure that results from the fact that the successor to a trapped Rule 90 configuration, with all live cells falling within a subset defined by every fourth cell, is always a similarly Rule 90 configuration. The pressure may be strong but it is not insurmountable as the presence of one period 5 loop at trap size 10 illustrates.

Even on my aging G4, collecting this form of statistics proved practical up to width 23 where that largest exhaustive run is recorded as having taken 48607 seconds. But the number of trap patterns keeps doubling, so the only answer was to start randomly sampling which meant data like longest "hair" instantly lost continuity, as quickly did the frequency count for smaller loop sizes.

The size of loops and the length of "hair" also grows exponentially, though still a lot slower than the total number of possible patterns, so even a small sample starts to take significant time at larger trap sizes. *Trapper* also encounters progressively larger "monster" loops at larger sizes which presented quite an obstacle until I found a method to deal with them. Eventually the sample sizes ranged from a million for width 24 down to ten for width 130 before I decided enough was enough. An extract of the consolidated data shows the ranges sampled at each sample size:

```
Sample: 1000000
24:   Loops: 252x2 56x3 55 31x2 28x3 26 22x2 12x4 5x3 4x75 (die:9548, branch:990356, 6992
secs.)
37:   Loops: 2044 252x10 48x65 28x9 26 24x8 21x2 13x2 12x13 5x2 4x14 (die:1429,
branch:998444, 73411 secs.)
Sample: 100000
38:   Loops: 260 252x16 48x54 28x13 24x6 21x4 12x6 5 4x5 (die:790, branch:99104, 9146
secs.)
55:   Loops: 65532 1182 124x261 120x1188 112x144 60x51 56x4 30x4 28 24x37 12x15 8 (die:467,
0:35503, branch:62323, 62279 secs.)

Sample: 10000
56:   Loops: 65532x2 461 124x182 121 120x1058 60x42 30x2 24x9 12x8 4 (die:207, 0":3096,
0':3188, branch:2205, 7473 secs.)
76:   Loops: 16380x252 5460x12 504x12 96x4328 48x25 24 (branch:5370, 67612 secs.)
Sample: 1000
77:   Loops: 16380x95 5460x2 4092x2 504x5 96x735 (9077 secs.)
99:   Loops: 8184x329 1020x549 1008x110 (branch:12, 86676 secs.)
Sample: 100
100:  Loops: 1020x68 1008x32 (9629 secs.)
112:  Loops: 2044x96 240x4 (36958 secs.)
116:  Loops: 2147483644x2 248x2 240x58 124 (0":20, 0':19, 45373 secs.)
121:  Loops: 4294967292 252x22 248x33 124x3 (die:26, 0:16, 79447 secs.)
Sample: 10
113:  Loops: 131064x4 2044x2 240x4 (3981 secs.)
115:  Loops: 2147483644 131064x7 240 (0:2, 10924 secs.)
122:  Loops: 252x5 248x2 (die:3, 9614 secs.)
130:  Loops: 248x10 (31637 secs.)
```

The extra frequency counts often immediately preceding the other branch frequency and/or the run seconds figure have to do with "monster" loop detection. The relatively reduced sample size for width 113-115 is due to a particular issue with their size 131064 loops, the resolution of which has been left until after a planned refactoring of the whole process to retain more comprehensive data for each *Trapper* seed pattern tested. Loop size 2^{32} - 4 = 4294967292 at width 119-121 is the biggest identified, but an even larger loop is known to be lurking at width 131 which made width 130 a good place to pause.

As much as the detection and handling of monster loops has been a very interesting part of the *Trapper* story, I have found it also particularly interesting that in addition to all the now mostly predictable families of loops with periods that are multiples of four,[25] a much smaller portion of other odd loop sizes continued to be detected, the largest so far being 3341. There appears to be no clear pattern to the first appearance of such sizes and their existence appears to reflect the random chance that one of the ever growing "hairs" will loop back to itself before it gets onto the slippery slope to Rule 90 territory.

1.1.19 Appendix D: Philosophical Implications

My main motivation for this kind of enquiry is to see how simple mechanisms in the subatomic microstructure might explain and constrain the macroscale universe that we find ourselves born into. My methodology is consistent with the program expounded in 2002 by Wolfram as *A New Kind of Science.*[see 7]

Each of the experiments discussed here has provided hints of initially surprising possibilities:

- While intended only to be an oversimplified test bed for experimenting with one popular candidate mechanism for the finest microstructure of the universe, *Tick Tock* has already shown that even such a simple mechanism can give us something analogous to inflation at no extra cost.
- The laws of microscale physics are generally formulated so as to appear invariant under time reversal, leading to the question as to how macroscale irreversibility emerged from microscale reversibility. *Trapper* suggests this might be getting things backwards, clearly demonstrating the unavoidable and reasonably rapid evolution of irreversible microscale systems into cyclic and thus reversible systems at the next larger scale.[26]
- Wolfram's identification of Class 4 CAs mentioned in the background section on complex systems theory[see 3] and other work in the 1980s identified the "edge of chaos–border of order" as the place to look for the origins of the complex organisation that we find in our universe. Follow up studies found that border elusive and started something of a retreat. However results from *Life in a Tube* in particular continue to suggest to me that it is still the best place to look.[27]
- Atomic theories seem to carry an implicit but unnecessary assumption that when you have found the real atoms there will be nothing left over to be explained. This is of course now being strongly questioned in the biological world with ever increasing evidence that genes do not tell the whole story. Yet even in this age of preoccupation with "dark matter" there is no sign of anybody thinking about the possibility that the microscale mechanism which produces familiar matter may at the same time be producing unrelated things. The odd loops in *Trapper* do not fit into any predictable families but that does not make them any less real.
- My original *Pattern Breeder,* mentioned in the introductory section on cellular automata, especially when viewed from 2004, hints at the possibility of a reversible, and thus zero cost, mechanism for replicating a seed pattern across all of space. An analogous mechanism could conceivably underpin Rupert Sheldrake's widely unpopular notion of morphic resonance.[28]

In a couple of these cases I started with certain hopes and expectations. In others the results came as quite a surprise, at least until I had time to reflect more deeply. Even after that there remain a couple of points revealed by the data which go against the grain of some long held beliefs. That alone is good reason to keep thinking about the next interesting experiment that Perl might make practical.

1.1.20 Notes

[1] Conway's Life is described in great detail at http://www.ericweisstein.com/encyclopedias/life/

[2] In AK Dewdney, Wallpaper for the Mind, Computer Recreations, *Scientific American,* September 1986

[3] Wolfram's 1983 paper is reproduced at:
http://www.stephenwolfram.com/publications/articles/general/83-cellular/

[4] *A New Kind of Science* p. 231 ff, see also note 7.

[5] Wolfram has long had ultimate responsibility for the journal Complex Systems:
http://www.complex-systems.com/

[6] http://www.wolfram.com/products/mathematica/

[7] Information about the NKS book and related developments can be found at http://www.wolframscience.com/

[8] Planck scale implies of order 10^{43} ticks per second. Both Wolfram in NKS (note 7) and, from a quite different perspective, quantum gravity theorist Lee Smolin see such a network as a likely candidate for the ultimate microstructure of our universe.

[9] The pentatope is easily mapped onto 3-D space as a tetrahedron with all vertices connected to a central point and even into 2-D space as a pentagon with each vertex also joined to its 2 opposite vertices forming an internal 5 pointed star.

[10] Starting with the forum software used at http://www.transforum.net/ and client sites.

[11] As per my lightning talk based on http://www.meme.com.au/OSDC/WhyPerl.pdf

[12] The *Tick Tock* website is at http://www.twistet.com/

[13] C.f. my lightning talk about using 3 OS X Macs as staging servers for developing Mod_Perl-based Linux-hosted http://www.miettas.com.au/ in particular.

[14] http://www.barebones.com/products/bbedit/index.shtml

[15] Also used to produce http://www.meme.com.au/OSDC/wheel.gif for another lightning talk.

[16] Each of the joined tetrahedra produces a direct successor at the next tick. Their joining edge is shared by four triangles and generates another tetrahedron that shares opposite edges with each of the other two. This process of doubling the joining edges is repeated each tick.

[17] I am very keen to find a modern replacement which displays 3-D wireframes without embellish-ments. ChemRote is also still used by: http://jcrystal.com/steffenweber/POLYHEDRA/p_00.html

[18] An efficient n+1 dimensional coordinate system used to represent n-simplexes is discussed at http://www.twistet.com/ticktock/coord_frames.html

[19] Some simplexes may alternatively be joined at a single vertex but single vertex joins do not persist for even another tick.

[20] A similar Perlish pseudocode version of the second generation algorithm is presented for compar-ison with the one shown earlier in this paper at http://www.twistet.com/ticktock/algorithms.html

[21] http://www.tcf.vt.edu/systemX.html

[22] A couple of largish animations of more complex outcomes are available as downloadable attachments to my follow up posts in an NKS Forum thread where I presented some results from *Life in a Tube:* http://forum.wolframscience.com/showthread.php?postid=410

[23] LifeLab has the extra bonuses of running various 2D and 1D rules, the ability to tile an arbitrarily sized universe with patterns of any size, lots of controls and the use of the RLE plain text format for saving patterns: http://www.trevorrow.com/lifelab/

[24] I posted an earlier progress report, accompanied by another simple yet interesting animated GIF to TransForum in June 2003: http://www.transforum.net/m.cgi?num=215

[25] These unpredictable loops do not actually avoid sizes which are multiples of four, rather the predictable families have sizes which are such multiples.

[26] My first, October 2003, contribution to NKS Forum was "Irreversibility precedes reversibility": http://forum.wolframscience.com/showthread.php?threadid=78

[27] http://www.meme.com.au/theoria/retreat.html

[28] http://www.sheldrake.org/papers/Morphic/