A search for plane-filling fractal monster curves on the triangle-grid

December 3, 2018

Abstract

Plane-filling curves have been around since 1890, but up to 2013 only relatively few examples of such objects have been known. Our effort for determining all curves of certain kinds were stopped by the combinatorial explosion inherent to the problem. Parallel computations funded by KONWIHR will allow us to extend the search considerably.

We developed a new algorithm from scratch, hoping to gain a speedup even for computations on a single core. Our hope was for an improvement by at least a factor of ten compared to the previous software. Turns out, our new implementation is (at least) half a million times faster! We describe key aspects of the algorithm and implementation.

1 Introduction

A curve on the triangle-grid is shown in Figure 1. The shape of the curve exhibits self-similarity, it can be decomposed into 13 smaller copies itself as indicated by the colors.

This curve can be rendered as follows. The *motif* of the curve, consisting of 13 edges (of unit length) is shown on the left of Figure 2. Replacing each edge with the motive itself gives what's shown on the right. Repeating this process of edge-replacement a few times gives the curve in Figure 1.

Note that the curve is self-avoiding, it never crosses itself. It also is plane-filling, covering the edges of arbitrarily large parts of the plane.

A necessary and sufficient condition for a motif to actually give a plane-filling and self-avoiding curve can be stated surprisingly simple. Draw three copies of the motive in both a clockwise and a counter-clockwise fashion. We call those arrangements *tiles*, as they actually tile the plane. If the edges in the interior of both tiles are completely covered and no crossing occurs, we have a valid curve (see [3]).

Our previous search essentially generated all three possible turns (by 0, +120, or −120 degrees) between each pair of adjacent edges and checked for the tile-condition. Obviously this algorithm is $O(3^R)$ where R is the number of edges, optimistically assuming that checking the tiles is $O(1)$. The search for $R = 31$ took about 4 days. The next R where one does find any curve is $R = 36$ and the search would take between one and two years on a single core.

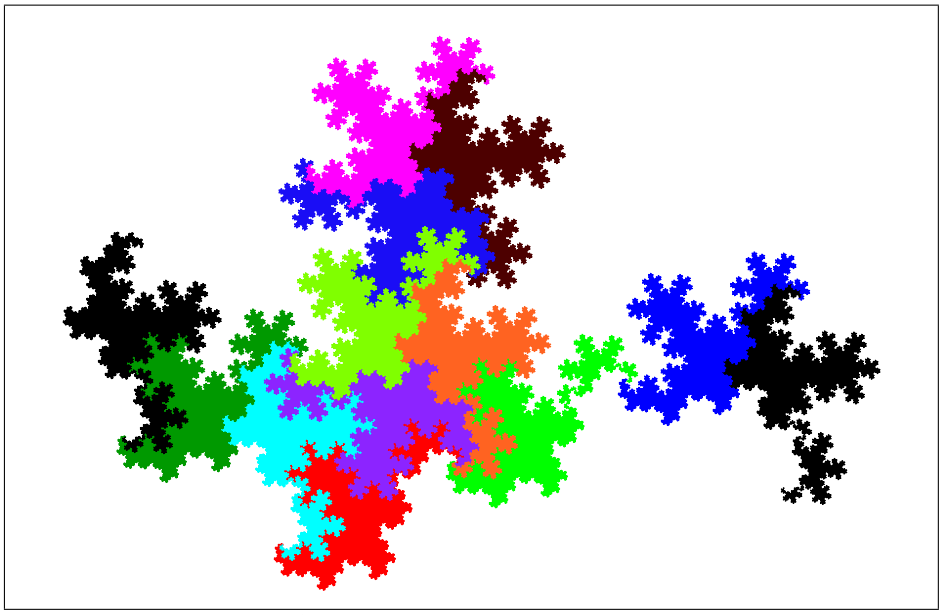

Figure 1: A plane-filling curve on the triangle-grid.

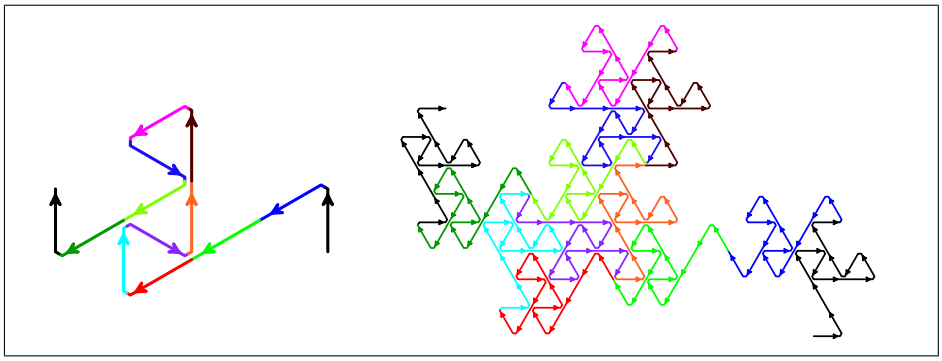

Figure 2: The motif of the curve (left) and the curve obtained by replacing every edge of the motif by the motif itself (right).

2 Algorithm and implementation

Each R where curves can be found is necessarily of the form $R = x^2 + x\,y + y^2$ with $x, y \in \mathbb{N}$, where the choice for x and y may not be unique. We need to examine all such pairs (x, y) to fix the end-points of the motifs (starting from the origin). For each such choice of start- and end-point, all six starting directions (from the origin) have to be considered.

We use a recursive algorithm. From the end-point of a prefix of the motif, try all points reachable with each of the three turns allowed. As any point is tried, a configuration of motifs equivalent to the two tiles is incrementally drawn. Every time the end-point is reached, we have found a new curve.

Memory access considerations. For each grid-point we only need to store six bits of information, one for each adjacent edge, using one `unsigned char`. Three 2-dimensional arrays are used: the first for one copy of the curve alone, the second for the arrangement of curves shown in Figure 3, the third holding (in an `unsigned short`) for each point how many steps it takes to reach the end-point.

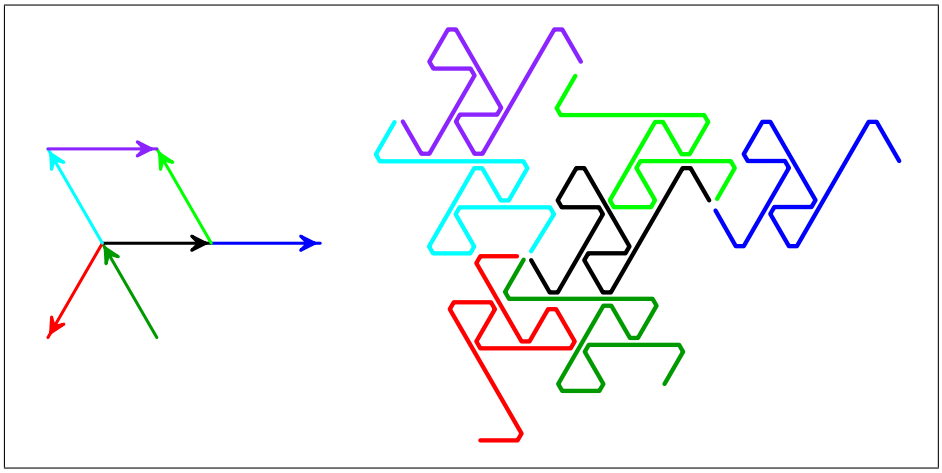

Figure 3: Outline of the arrangement of curves used in our search (left), and partially finished curves drawn corresponding to the outline (right).

The arrays fit into first level cache, resulting in close to zero cache misses on a intel Xeon CPU (E3-1275 V2 3.50GHz) according to `perf stat`:

```
    13577.326810   task-clock (msec)         #    0.999 CPUs utilized
  47,295,453,264   cycles                    #    3.483 GHz
  13,984,619,046   stalled-cycles-frontend   #   29.57% frontend cycles idle
 107,678,299,554   instructions              #    2.28  insn per cycle
                                             #    0.13  stalled cycles per insn
  10,093,672,424   branches                  #  743.421 M/sec
     277,367,047   branch-misses             #    2.75% of all branches
  41,028,083,667   L1-dcache-loads           # 3021.809 M/sec
       2,427,138   L1-dcache-load-misses     #    0.01% of all L1-dcache hits
```

On a more recent CPU we reached 2.80 `insn per cycle`.

For better memory locality the three arrays are folded into one, using an array of `struct {point; point; min_distance;}`.

The first fully working version of our implementation used a Z-shaped arrangement of only three curves (shown in dark green, black, and light green in Figure 3) and was already faster by a factor of 1000 over the previous program. What followed was a four-week effort to get the best possible performance. We highlight a few crucial considerations and techniques.

Skipping starting directions. Of the six possible starting directions a few can usually be skipped altogether. Either because there is no curve starting in that direction (a parity condition), or all resulting curves are duplicates of others with another start-direction. This saves between 1/3 and 2/3 of the search effort.

The omega-rho problem. A prefix may enclose an empty part of the grid that cannot possibly be visited in the future (omega configuration) or enters

a part of the grid in a way making it impossible to leave (rho configuration). Firstly, we needed to detect when this may just be happening, namely when exactly two (circularly) adjacent bits are zero. As $2^6 = 64$ we can use a single word as a lookup table:

```
bool cell_may_close_loop(unsigned char z)
{
    static constexpr unsigned long m =
    ( 1UL << 0b001111UL ) |
    ( 1UL << 0b011110UL ) |
    ( 1UL << 0b111100UL ) |
    ( 1UL << 0b111001UL ) |
    ( 1UL << 0b110011UL ) |
    ( 1UL << 0b100111UL );
    return  ( ( (m >> z ) & 1UL) == 1UL );
}
```

Techniques like this are used in various places of our program, see [4] or [1] for rather extensive collections of such tricks.

To determine whether there actually is a problem, the area enclosed by the loop is computed using the formula $A = \frac{1}{2} \sum_k x_{k1}\, y_{k2} - x_{k2}\, y_{k1}$ where the sum ranges over all edges of the loop and (x_{k1}, y_{k1}) and (x_{k2}, y_{k2}) are respectively the coordinates of the start and end point of the kth edge. That area A must match the number of edges on its border (up to a constant factor). Floating point numbers are used for the area calculations, because the computations run essentially in parallel to all other ones using only integer operations.

The mid-point problem. Sometimes a grid point lies on the mid-point of the tile. If so, when three curves meeting there take a turn in the wrong direction, none of them can reach its end-point. Therefore we detect if the point visited is a mid-point and suppress the wrong turn. This issue alone can lead to a slowdown by a factor of ten if not addressed.

Everything needed to be perfect. In the course of fine-tuning the program we noticed that while sometimes the rate of finding curves was very good, there sometimes was a "pause" where zero curves were found. Incremental improvements did not avoid the issue to our satisfaction.

Indeed, only after adding the last curve in the arrangement shown in Figure 3, the rightmost arrow, that the issue was solved. This led to the final major speedup, from a factor of two up to 100, depending on the starting direction. The overall improvement being at least by a factor of ten.

3 Parallelization

Now we had a problem. Due to the unexpected rate of finding curves, we generate about 30 Gigabytes per hour per core!

That amount can be cut down by a factor of about 1000 by discarding curves whose shape (configuration of edges) duplicates a shape already found, see Figure 4.

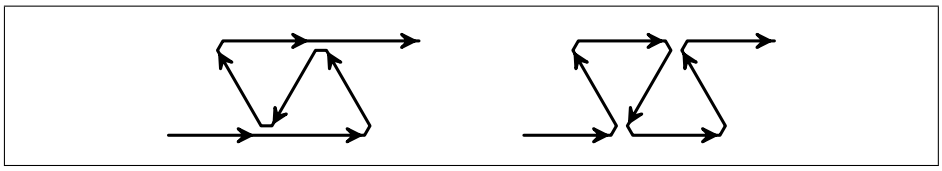

Figure 4: Two curves having the same shape.

This reduction step was planned to be done only on the complete output for each search, after the parallel search. Appending the step after each partial search leads to incomplete discarding of duplicates, making a final such step on the re-combined files necessary. However, the data reduction is just what we need.

The parallelization itself is conceptually easy. One version of our program, the "master" stops the recursion when a certain prefix-length is reached and prints a command for the other version, the "worker". The worker starts the recursion only after the prefix given to it in its arguments. No communication is required between the worker instances.

The curves obtained will be made available to the public as soon as we complete the computations.

Jörg Arndt and Julia Handl, Technische Hochschule Nürnberg

References

[1] Jörg Arndt: Matters Computational, Springer-Verlag (2011).

[2] Jörg Arndt: Plane-filling curves on all uniform grids, arXiv:1607.02433 [math.CO] (2016), http://arxiv.org/abs/1607.02433.

[3] Michel Dekking: Paperfolding morphisms, planefilling curves, and fractal tiles, Theoretical Computer Science, vol. 414, no. 1, pp. 20-37, (2012).

[4] Donald E. Knuth: The Art of Computer Programming, Volume 4, section 7.1.3: bitwise tricks and techniques, Addison-Wesley (2011).