VISUALIZING THE COLLATZ PROBLEM

It is relatively simple to explain the Collatz problem, in fact it is possible to understand the Collatz sequence only by looking at the first to lines of code. The more striking it is that it is still an open problem in mathematics. The problem is to show that every Collatz sequence reaches the value 1 in a finite amount of steps, regardless of the starting number.

Introductory Observations

```
Collatz[1] := {1};
Collatz[n_] := {n} ~ Join ~ If[EvenQ[n], Collatz[n/2], Collatz[3 n + 1]];
```

Here are some interesting examples of collatz sequences. The starting numbers 31 and 32 produce very different sequences, partially because 32 is a power of 2 and 31 is not. Notice also that

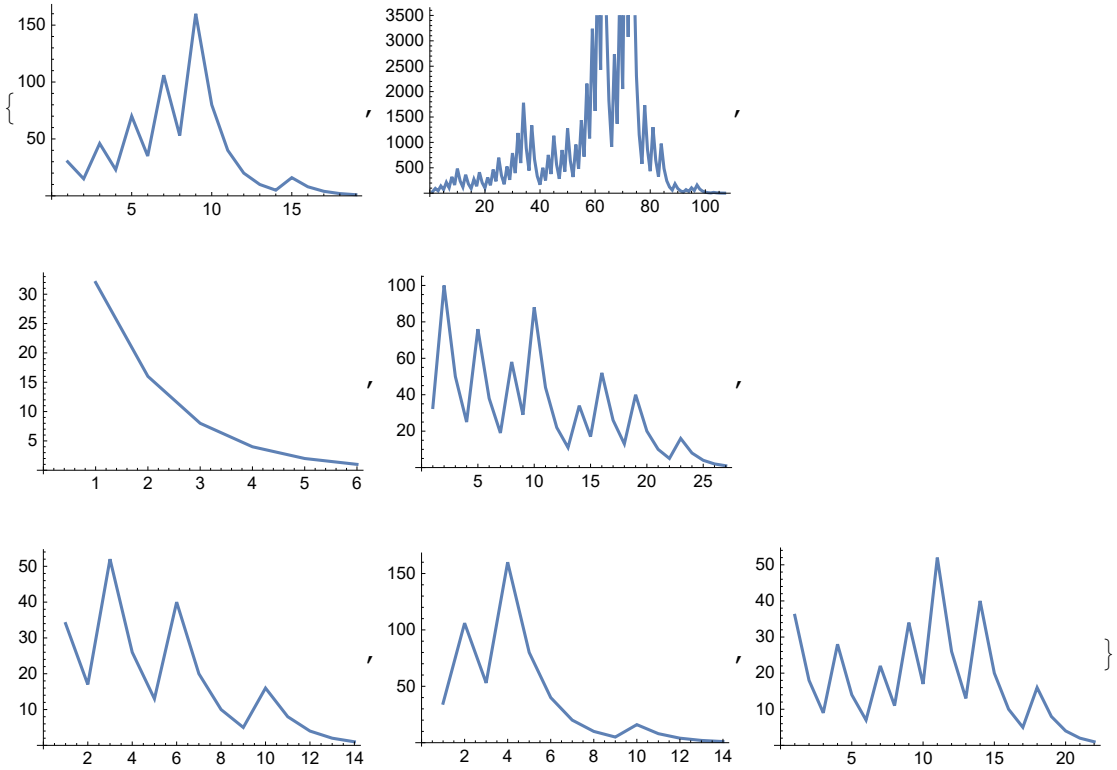
```
Collatz[15]
```

```
{15, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1}
```

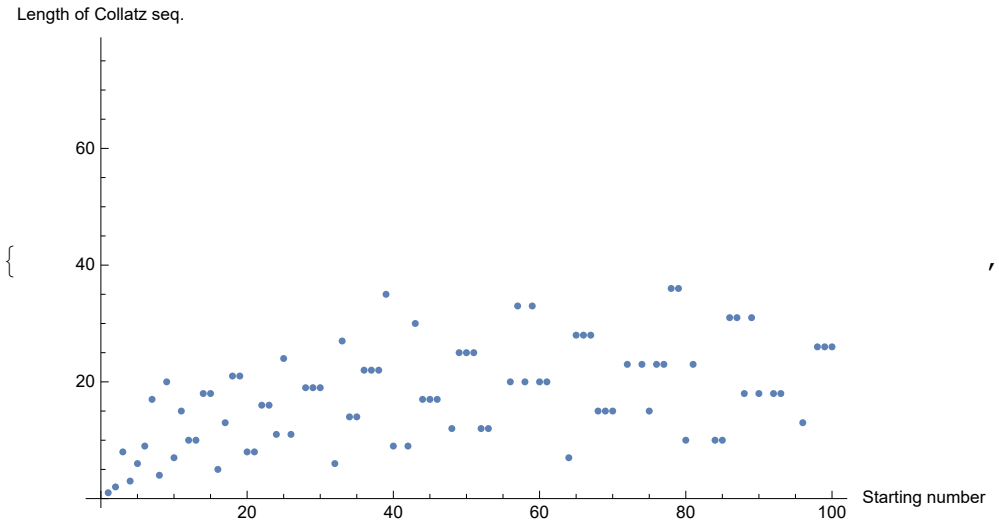
```
Collatz[31]
```

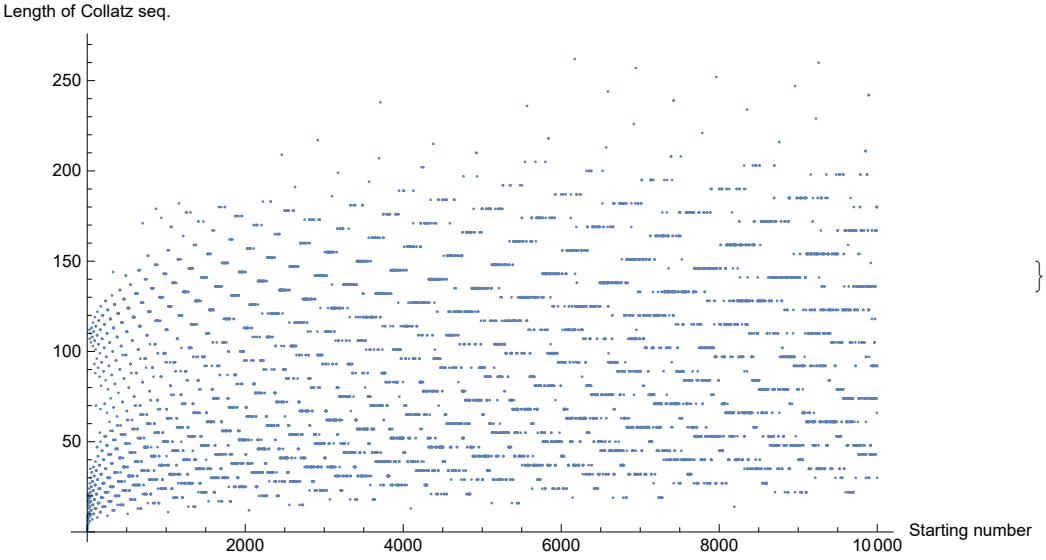
```
{31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274,
 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263, 790, 395,
 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566,
 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619,
 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232,
 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122,
 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1}
```

```
Table[ListLinePlot[Collatz[i]], {i, 30, 36}]
```



```
Table[ListPlot[Table[Length[Collatz[i]], {i, k}],
  AxesLabel → {"Starting number", "Length of Collatz seq."}], {k, {100, 10 000}}]
```





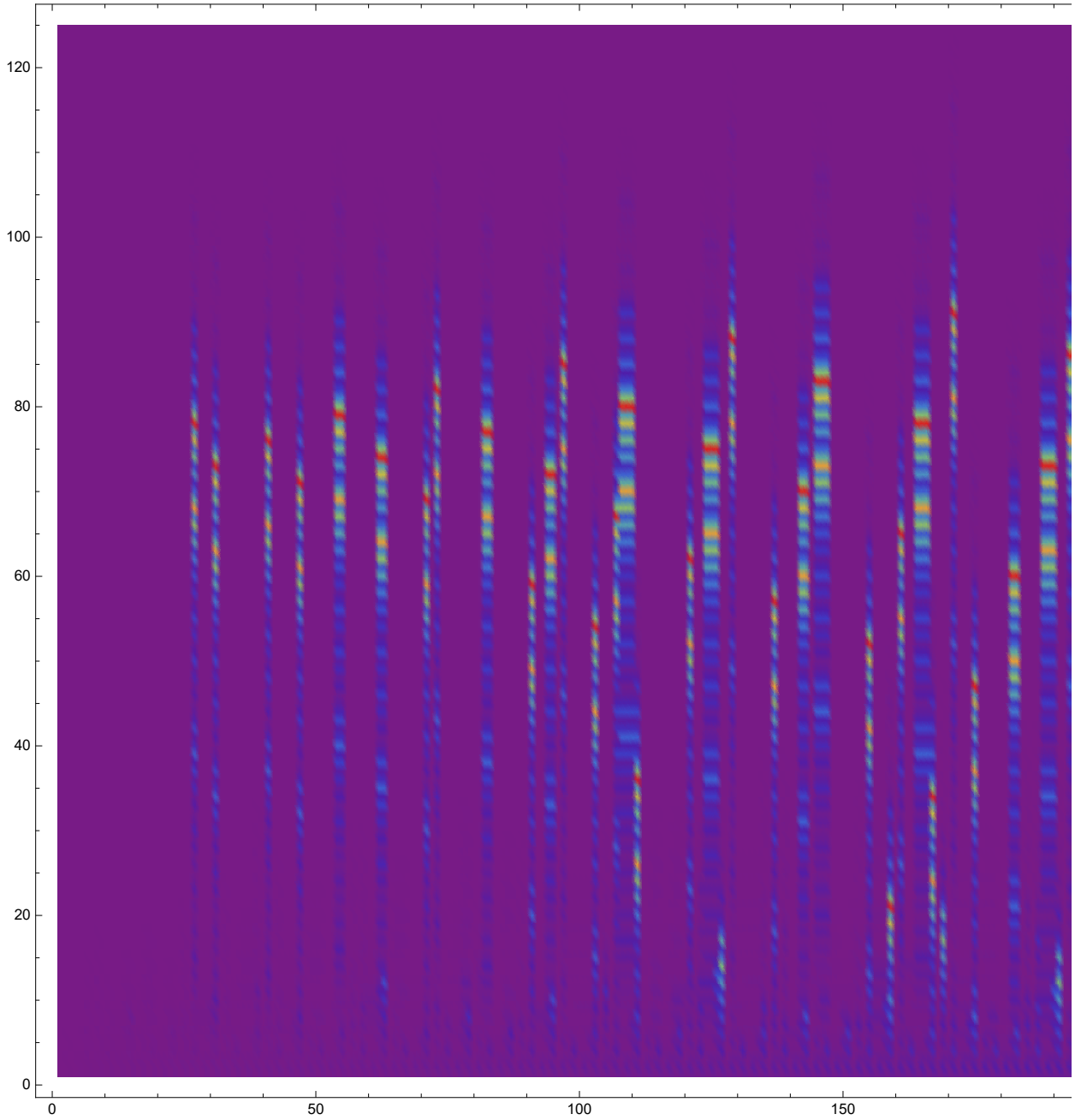

```
Do[Print[Collatz[i]], {i, 10}]
```

```
{1}
{2, 1}
{3, 10, 5, 16, 8, 4, 2, 1}
{4, 2, 1}
{5, 16, 8, 4, 2, 1}
{6, 3, 10, 5, 16, 8, 4, 2, 1}
{7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1}
{8, 4, 2, 1}
{9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1}
{10, 5, 16, 8, 4, 2, 1}
```

The Collatz sequence has, as we have noticed, not the same length for different starting values. So a list of Collatz sequences is not a well defined matrix. Therefore we need a function that adds zeros to every Collatz sequence in a list until every sequence has the length of the longest sequence in the list. Then we can use *ListPlottable* to convert the list of Collatz sequences in the required form, as indicated in the Wolfram Documentation Center, to create a *ListDensityPlot*

```
FillUpWithZeros[array_] := Module[{max},
                                  max = Max[Length /@ array];
                                  (# ~ Join ~ Table[0, {i, max - Length[#]}]) & /@ array
                                  ];

col = Table[Collatz[i], {i, 200}];

mat = FillUpWithZeros[col];

ListPlottable[mat_] :=
  Table[{i, j, mat[[i, j]]}, {i, Dimensions[mat][[1]]}, {j, Dimensions[mat][[2]]}];

plot = Flatten[ListPlottable[mat], 1];
```

```
ListDensityPlot[plot, PlotRange → All, Mesh → None,
 InterpolationOrder → 3, ColorFunction → "Rainbow"]
```



The above plot shows very well the different lengths of different Collatz sequences and also the peaks indicated by the color.

Going Forwards And Backwards

Assuming the claim that any Collatz sequence ends in 1 regardless of its initial value, one can reach

any natural number by going some Collatz sequence backwards.

This can be visualized best with a graph that connects n to $2n$ and if $n-1$ is a multiple of 3 also with $\frac{n-1}{3}$.

```
MultipleOf3[x_] := If[Mod[x, 3] == 0 && x ≠ 0, True, False];

CollatzBackwards[start_, m_] := Module[{a = {start}, last},
                        While[Length[a] < m,
                              last = Last[a];
                              If[Mod[last - 1, 3] == 0,
      AppendTo[a, Prepend[CollatzBackwards[(last - 1)/3, 5], last]];
                              AppendTo[a, 2 * last];
                         ];
                    a
                    ];

$RecursionLimit = 1 000 000;

c = CollatzBackwards[8, 10]
{8, 16, {16, 5, 10, {10, 3, 6, 12, 24, 48}, 20, 40},
 32, 64, {64, 21, 42, 84, 168, 336}, 128, 256,
 {256, 85, {85, 28, {28, 9, 18, 36, 72, 144}, 56, 112, {112, 37, {37, 12, 24, 48, 96, 192},
    74, 148, {148, 49, {49, 16, {16, 5, 10, {10, 3, 6, 12, 24, 48}, 20, 40},
       32, 64, {64, 21, 42, 84, 168, 336}, 128}, 98, 196,
     {196, 65, 130, {130, 43, {43, 14, 28, {28, 9, 18, 36, 72, 144}, 56, 112}, 86,
       172, {172, 57, 114, 228, 456, 912}, 344}, 260, 520}, 392}, 296}, 224}, 170,
  340, {340, 113, 226, {226, 75, 150, 300, 600, 1200}, 452, 904}, 680}, 512}
```

This information could now be used to plot a graph where the natural numbers are vertices and edges connect verices of numbers which are connected by the Collatz sequence in terms of being successive numbers in a sequence.

Even though this Module works correctly, it is exceeding the recursion depth 0f 1024 very quickly, even setting a new recursion limit doesn't help a lot. The recursion depth is exceeded that quickly, because the function calls itself whenever there is a junction. The next function uses therefore a different approach that does not rely on recursion, but cycles internally and creates the graph iteratively.

The following function starts with the number 1 in a list *cur* of all values which are currently at the very "front" of the graph and connects each of them to twice of itself. The function also adds to the set *link* an undirected edge between each of those numbers and double this number and collects all new numbers which are now representing vertices at the very "front" of the graph in a list *new* which is going to be the new iteration of *cur*. Now everything repeats itself. But this would simply give straight lines starting each at one element of the initial list *cur*. So to to speak a real line bundle over the discrete set *cur*. To make things more interesting the function also checks of the elements e in *cur* are such that 3 divides $e-1$. In that case it also adds an undirected edge $e \longmapsto \frac{e-1}{3}$ and adds the latter one to the new iteration of *cur*.

```
CollatzGraph[n_] := Module[{cur = {1}, new = {}, links = {}},
                            While[Length[links] ≤ n,
                                  (*Print[Sort[links]];*)
                                  Do[
                                       AppendTo[links, e ⟼ 2 e];
                                       AppendTo[new, 2 e];

    If[Mod[e - 1, 3] == 0 && OddQ[(e - 1)/3] && (e - 1)/3 ≠ 1, AppendTo[links, e ⟼ (e - 1)/3];
                                                               AppendTo[new, (e - 1)/3];
                                       ];
                                   , {e, cur}];
                                  cur = new;
                                  new = {};
                             ];
                        links
                   ];
```

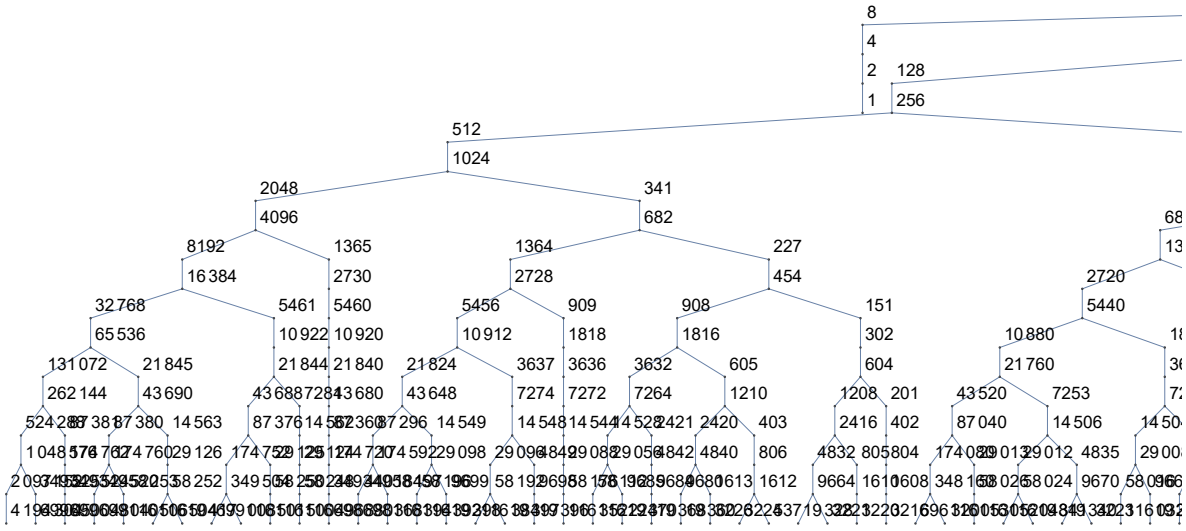
With the right *GraphLayout* this function creates beautiful graphs in which to the number of every vertex the collatz sequence is nothing else than the sequence of numbers of the vertices that lie on the path from the initial vertex to the very center, which is 1.

```
g = Graph[CollatzGraph[10 000], EdgeStyle → Dashed,
  VertexStyle → Black, GraphLayout → {"PackingLayout" → "ClosestPacking"}]
```

To make this plot nicer, I copied the *ReproduceGraph* function from the Visualize Knot project which creates a graphics object corresponding to any given knot. The can be added, since the Length of the Collatz sequence of the vertex number grows linearly outwards.

```
ReproduceGraph[g_Graph] := Module[{v, a, d, l, m, n, b, c},
                                  color[n_, k_] := (ColorData["Rainbow"][k/n]);

   v = Round[ VertexCoordinates /. AbsoluteOptions[g, VertexCoordinates], 0.1];
                                  a = AdjacencyMatrix[g];
                                  d = Dimensions[a][[1]];
                                  l = VertexList[g];
                                  m = VertexCount[g];
                                  n = Length[Collatz[l[[m]]]];

   b = Flatten[Table[If[a[[i, j]] ≥ 1, {color[n, Length[Collatz[l[[i]]]]],
        Line[List[v[[i]], v[[j]]]]}, {}], {i, d}, {j, d}], 1];
                                  c = DeleteCases[b, {}];
                                  Graphics[c]
                                ];

ReproduceGraph[g]
```

```
Graph[CollatzGraph[500], VertexLabels → "Name",
 GraphLayout → "LayeredDrawing", VertexSize → Tiny]
```

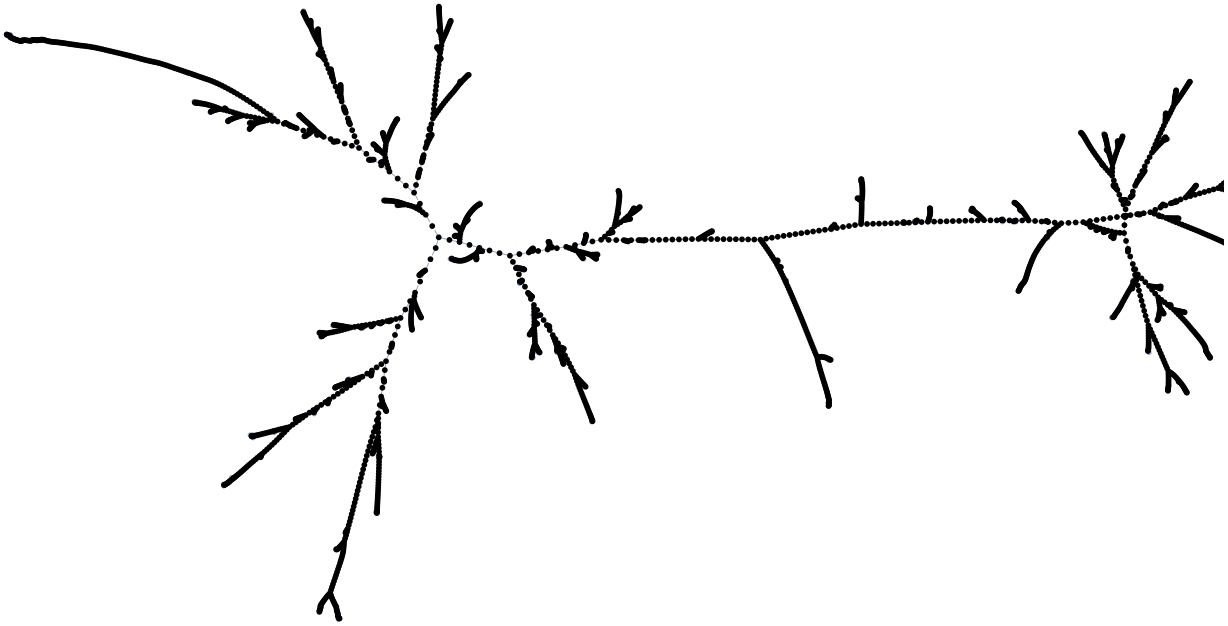


Another *GraphLayout, as shown* above, gives a very good intuition that the claim of the Collatz problem might actually be true since both strongly suggest that one can fill the entire set of natural numbers with the numbers corresponding to vertices in this graph. The upper plot of the graph bears striking resemblance to the veins in a lung.

This way of creating the Graph backwards, doesn't miss any "path" or junction. Another possible approach is to go the "right" direction, namely forwards and not backwards, and connect consecutive numbers in a Collatz sequence with an undirected edge, which the following function is doing. As one can see, the following graphs contain by construction of the algorithms not as many short branches, but rather fewer longer branches. In both graph the more red an area of the graph is, the longer are the Collatz sequences of the vertex numbers and the more blue an area is, the shorter are the Collatz sequences. 1 has clearly the shortest Collatz sequence.

```
CollatzForwardGraph[n_] := Module[{a, done = {}, links = {}},
                                  a = Table[Collatz[k], {k, 2, n}];
                                  Do[Do[
                                            If[

        ! ContainsAll[links, {p[[q]] ↦ p[[q + 1]]}],

        AppendTo[links, p[[q]] ↦ p[[q + 1]]];
                                                          AppendTo[done, p[[q]]];
        AppendTo[done, p[[q + 1]]]
                                                ];
                                    , {q, Length[p] - 1}], {p, a}];
                                  links
                                ];
```

```
h = Graph[CollatzForwardGraph[1000], EdgeStyle → Dashed,
  VertexStyle → Black, GraphLayout → {"PackingLayout" → "ClosestPacking"}]
```



```
ReproduceGraph[h]
```