Comp 1130

Assignment 1 – Wiring up Wireworlds

- by -

Nick Sifniotis  
u5809912

First draft

If you are reading this message, I was not able to submit my final report on time.

The sources for problems 1 and 2 are working, and the binary counter extension to Wireworld is included in this package as well.

Executive Summary

Solutions to the Wireworld problems were developed iteratively, and a collection of development highlights are presented in the sections for Problem 1 and Problem 2.

The optimal Problem 1 solution was based on a radix trie data construction. This solution achieved linear time complexity with respect to the size of the world but it was unable to attain constant time complexity as the number of heads in the world increased.

The optimal Problem 2 solution was based on a perpetually decapitated list-triplet construction. This solution was found to be both linear with respect to the size of the world, and constant time with respect to the number of heads within the world. While it is not as fast as the optimal solution to Problem 1, it is a more consistent algorithm that does not deteriorate as the number of heads increases.

Source codes from various stages of development, the data collected on the runtimes of the various iterations of the solution, and various other notes and errata can be found in the appendices. They are not an essential part of this report and are provided to satisfy the reader's curiosity[[1]](#footnote-2).

The Wireworld binary counter extension was successfully programmed. The input and output bitmaps are already attached to the structure. Please note that when testing it, the input electrons will need to be synchronised with the internal clocks of the counter structure. To assist in keeping the input synchronised, markers have been placed along the input wire indicating where electron heads can be placed. *The author cannot guarantee that the binary counter will function as intended if electron heads are placed incorrectly on the input wire*.

Problem 1

The programming task in problem 1 was to efficiently implement the function transition\_world using the List\_2D data structure that had been provided. The final solution was developed iteratively, and it passed through a number of different stages. The most interesting stages of development are presented below.

# Mark 1 – Naive solution

The most immediate and obvious solution was to utilise the existing functions that had been provided within Data.For\_List2D. The transition function was implemented by recursively processing each element within the 'world' data structure. Whenever a conductor cell was encountered, the number of neighbouring cells that were heads would be calculated by using the function element\_occurrence, which was one of the functions provided in that file.

The algorithmic complexity of this solution is O(n squared) with respect to the size of the dataset, because the recursive journey through 'world' is O(n) and each time a conductor is encountered, element\_occurence would itself traverse 'world' again in its search for Head cells.

This approach got Wireworld working, however it did not scale up very effectively. It took Wireworld over nine seconds to process the default 25 transitions on Langton's 3x3. That is almost a thousandfold increase in the runtime of the program, for a file that is only six times larger. No further testing was conducted on this solution.

|  |  |
| --- | --- |
| # of transitions | Seconds (testing the default bitmap) |
| 25 | 0.198s |
| 100 | 0.800s |
| 250 | 2.147s |
| 500 | 4.113s |

The raw data used to derive these figures may be found in the appendix.

# Mark 2 – No more empties

An immediate improvement in performance was achieved through the realisation that Empty cells, which always transition to Empty cells, can be safely removed from the simulation. One line of code was all it took to effect this improvement, and that line has been highlighted in the source codes that follow this report.

This modification has the effect not of altering the algorithmic complexity of the function, but of simply decreasing the size of the dataset that it is being asked to work on. The function is still, sadly, O(n squared). However, there was a reduction in the average runtime of the function of about 15 – 25%.

Langton's 3x3 took 6.812 seconds to process 25 transitions, so no further testing was conducted on this solution.

|  |  |
| --- | --- |
| # of transitions | Seconds (testing the default bitmap) |
| 25 | 0.172s |
| 100 | 0.632s |
| 250 | 1.564s |
| 500 | 3.098s |

# Mark 3 – Counting neighbouring heads

Clearly, an O(n squared) algorithm was not going to do the job for any of the larger Wireworlds. A better algorithm had to be found. Ideally, that algorithm would run with O(n) complexity, which would mean that the number of times a cell in 'world' was touched would not depend on the number of cells – of any sort – that were present within 'world'.

What was causing the n squared complexity? The call to element\_occurrence, which as noted above, operates in O(n) time. Searching through the entire world from go to whoa, to find a collection of eight cells that sit immediately adjacent to a particular cell, seemed wasteful.

That method was replaced with a new algorithm that would iterate through the entire world once, and whenever a head was found, it would add the coordinates of its neighbours to an unordered list of such coordinates. That list represents the number of head neighbours a cell has, as each neighbouring head would update that list. Once the list is created, the world would be iterated through again, and whenever a conductor was found, the list of coordinates would be traversed to see how many, if any, neighbours of that cell are heads.

This had the effect of reducing the algorithmic complexity to O(n) with respect to the size of the world, as the 'world' was only being traversed twice. Of course, when talking about complexity, O(2n) == O(n).

The differences in the runtimes were quite marked. This algorithm was such an improvement over the first two that testing was conducted against Langton's 3x3 as well.

|  |  |  |
| --- | --- | --- |
| # of transitions | Seconds (default bitmap) | Seconds (Langton 3x3) |
| 25 | 0.010s | 0.040s |
| 100 | 0.087s | 0.190s |
| 250 | 0.213s | 0.651s |
| 500 | 0.459s | 1.833s |

100 transition benchmarks were also taken on Langton's 5x5, 7x7 and 11x11. These results are summarised in Figure 1. Clearly, the runtime of this algorithm is very O(n) with respect to the size of the worlds.

This is a gigantic improvement over the first attempt to solve Problem 1, and the solution certainly appears to be of linear complexity. However, the solution is *not* linear with respect to the number of Head cells in 'world'. As more Head cells are produced in the simulation, the length of the unordered 'neighbouring cells' list grows, and this array needs to be iterated from start to finish for each and every conductor cell in the world. In the worst case, where exactly half of the world's cells are Heads and the other half Conductors, this improved algorithm does no better than the naïve solution.

This solution was tested against the Seven Digit Display Wireworld, with the following startling results.

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Transitions | 25 | 100 | 250 |
| Number of Heads | 13 | 137 | 799 |
| Runtime (sec) | 0.056s | 1.451s | 35.370s |

While the solution was definitely an improvement on the naïve approach, it was not linear with respect to the number of heads in a Wireworld. As the number of heads varies throughout the simulation, it was important that a way be found to improve, yet again.

# Mark 4 – A trie by any other name

The key to making the function work with linear complexity was to find a data structure that allows the programmer to associate a value (the number of neighbours that are Head cells) with an index (the coordinate of that cell) in constant time. There are a few data structures that could do the job. An array would have been ideal, but unfortunately implementing an array-based solution in Haskell is beyond the scope of the author's abilities. Likewise, a hash table could not be implemented, as the basis of such a table is an array.

The data structure, then, had to be constructed either from lists or from trees. Not seeing any way by which a list-based data structure could solve this problem, the author looked to a tree based structure that the author believed to have been called a 'radix trie'.

In researching radix tries for the purposes of this report, it became apparent that the data structure that has been implemented to solve the Wireworld problem is not a 'true' radix trie. However, as the author has been calling it a radix trie for the past few weeks, it will continue to be so called in this report (albeit, erroneously).

\*\*\*\*\*\*\*\*\*\*\* insert description of radix trie here \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Radix tries – what are they – a data structure, based on a tree, that allows a programmer to associate a data element with an index – in this case, a number.

what is the algorithmic complexityy of each operation (o(1)!) with respect to the quantity of data stored, O(logn) with respect to the size of the indexes being used.

How do they work – each node has the same number of child nodes. This number is called the radix of the trie. Data is stored in a node by taking (key modulo radix) and storing it in the child node that corresponds to that value – the remainer of the key (key div radix) is then passed to that child as the new index, and so on it goes. If the node receives a key of zero, then it knows to store the data value in itself instead of its children. It's recursive!

## Implementation Details

The most basic implementation of this trie structure assumes that the index is positive.

A way needed to be found to map the two-number coordinate system as used by For\_List\_2D to an index that a radix trie can use. Combining the two numbers together, by multiplying the first by some constant such as 1,000,000 and adding it to the second, was considered. However it had the disadvantages of limiting the size of the Wireworlds that could be simulated – which the assignment specifications did not say could be assumed – and of forcing the program to make many large mod and div calculations even when the number of heads stored was very low.

A more general solution suggested itself. The radix trie data structure was designed to store any type of data, not necessarily an integer. It seemed perfectly natural, then, to store radix tries within some master radix trie. The x coordinate, then, indexes a radix trie that contains all the information that is stored for that 'line', and the y coordinate is used as the index for *that* radix trie to retrieve the desired value.

When this system was first tested, it produced problems straight away. The radix trie worked perfectly when it was tested manually but when it was tested against a Wireworld that was loaded from a bitmap, it would cause the computer to hang.

It was soon discovered, thanks to a late night post on Piazza and a crash course on the use of traceShow, that some of the coordinates that the Wireworld framework provides to the system are negative. Up to that point, the assumption had been that the framework loads the Wireworld data starting from the top left corner of the bitmap, and working rightwards and downwards in a positive direction. TraceShow put paid to that idea. The data structure had to be rewritten to accept coordinates that could be either positive or negative.

To solve this problem, a new data structure was created to wrap around the radix trie. Called FirstRadixNode, it contains two nodes that contain radix tries. Wrapper functions are used to test the sign of incoming indexes, and to return the radix trie that corresponded to that sign.

So the final data structure is a FirstRadixNode that returns a radix trie that returns a FirstRadixNode that returns a radix trie that returns an integer. O(1) time indeed.

As per Figure 1 above, the radix trie solution operates with a linear complexity, with respect to the size of the world. It also clearly outperforms the unordered list solution of Mark 3 above. In point of fact, this solution generated the fastest runtimes on Langton's 11x11 out of all the solutions in this entire assignment.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Wireworld | Langton's 3x3 | Langton's 5x5 | Langton's 7x7 | Langton's 11x11 |
| Runtime (s) | 0.112s | 0.267s | 0.510s | 1.210s |

This solution was also benchmarked against the Seven Digit Display Wireworld, to see whether or not it offered any improvement to mark 3 as the number of the heads in the world increased. Figure 2 shows that this is clearly the case.

However, when compared to the number of transitions instead of the number of active heads reported at the end of the program, the radix trie solution did not live up to its promise of offering O(1) indexation services.

|  |  |  |  |
| --- | --- | --- | --- |
| Transitions | 25 | 100 | 250 |
| Runtime | 0.029s | 0.270s | 2.413s |

The author can only assume that these results are an artefact of the constant reconstruction of the radix trie as data is inserted into it.

# Mark 5 – Really Bad Optimisations

Once the radix trie solution was operational, various methods of optimising the code to decrease the runtime further were considered. However, none of the optimisations were able to improve on Mark 4. Not a single one.

* The key to the radix trie is a called function split\_key, which returns a tuple containing 'this node's key' and 'the rest of the key'. This function was originally implemented using the mathematical operators 'div' and 'mod'. It was observed that the radix trie, as implemented, had four child nodes. As four is a nice, neat power of two, removing these mathematical operators and replacing them with bitwise operations seemed quite natural.

Haskell did not like being forced to perform ANDs and bitwise right shifts on its Integer types. This 'optimisation' added several seconds to the program's runtime.

* Radix tries are lightning fast in languages that allow variables and objects in memory to be changed. Unfortunately, Haskell is not one of those languages. Each and every time a new piece of data is stored in the trie, the entire trie has to be rebuilt with the new data inserted into it. There is a trade off between the cost of reconstructing the tree – particularly in terms of memory allocation – and the depth of the tree, which negatively affects the insertion and lookup costs. When the trie was first implemented in Mark 4, the nodes were programmed to have four children. This decision was one of convenience – it could well have been six, or two, or sixteen. There was no reason to believe that four was the sweet spot between the two competing forces of depth and memory usage.

Alternative tries were designed and tested against Langton's 11x11, on 1000 transitions. The surprising results may be found in Appendix I. Testing was not extensive, and while it is tempting to conclude that four children is the miraculous sweet spot, it may be more sensible to simply observe that a radix trie of four children per node seems to outperform radix tries of sizes two and eight.

* The processing and flow of control within the main transition\_world functions and its helpers was rebuilt from the ground up to minimise the number of times each cell had to be 'touched'. This was supposed to lower the runtime of the program, however, it had the opposite effect, inexplicably quadrupling the runtime of the program when tested against Langton's 11x11. The source code of this optimisation is included in Appendix II. It is not known why the runtime increased so markedly when the number of touches per cell was reduced. It could be because more lists are being created and passed around in recursive function calls. Or it could be that Haskell enjoys AFL and gives preferential processing time to player functions that generate more touches for the team.

Problem 2

Problem 2 is to implement transition\_world again, using a different data structure to the one presented in Problem 1. In this problem, the data structure is an ordered collection of an ordered collection of sparse lists.

The radix trie solution developed in Problem 1 has set the bar very high. Will it be possible to leverage the fact that the Ordered Lists are ordered and do better?

Mark 1 – Naive Solution

The first solution that suggests itself is the same as the naïve solution to problem 1; simply scan each line one at a time, and count the number of head nodes that neighbour every conductor cell. This was achieved by first creating a collection of triplets of lines – for every line in the data set, the triplet contained the line being scanned, the line immediately above it, and the line immediately below it. By grouping the lines in this fashion, the neighbourhood search space was cut down from the entire world to a set of three lines.

Benchmarks were taken against the four Langton worlds, and the results are below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Wireworld | Langton's 3x3 | Langton's 5x5 | Langton's 7x7 | Langton's 11x11 |
| Runtime (sec) | 0.646s | 3.068s | 8.782s | 36.421s |

The complexity of this algorithm is not known – it looks like O(n squared), and for every conductor cell that is found, three horizontal lines are traversed in the neighbourhood search. The runtime, then, is a function of the number of heads in the world and the length of the lines in the world. In any event, it is clearly not linear.

Mark 2 – Nearly Headless Lists

By cutting off the heads of the triplet-lists as they are scanned, it is possible to reduce the search space even further and get faster runtimes. This is as close to O(n) as it is possible to get, and it takes advantage of the fact that a cell's neighbours are located 'close' to the cell being examined.

The mark 2 solution is exactly the same as the naïve solution, with the exception that once a neighbourhood search is performed, any entries in the triplet-lists that come before the 'current position' of the cell being scanned are dropped. This takes advantage of the fact that the position of the cell being scanned is always increasing, a consequence of the fact that the lists are ordered. Every element in the triplet-lists will be touched at most two times. This is an improvement over the previous solution, where elements in the triplet lists could be touched up to k times, where k is the number of heads in the list being scanned.

As per the previous solution, this was benchmarked against the four Langton Wireworlds on 100 transitions.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Wireworld | Langton's 3x3 | Langton's 5x5 | Langton's 7x7 | Langton's 11x11 |
| Runtime (seconds) | 0.152s | 0.433s | 0.870s | 2.178s |

These results are about as good as those that the Mark 3 solution – pre radix trie – was able to achieve in Problem 1. However, further benchmarking of this solution revealed something quite interesting; this solution was of order O(1) with respect to the number of heads in the Wireworld. Benchmarking against the Seven Digit Display yielded the following results.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Transitions | 25 | 100 | 250 | 500 |
| Runtime (s) | 0.046s | 0.155s | 0.365s | 0.721s |

As Figure 3 shows, this solution clearly blows the radix trie approach right out of the water.

The explanation for this astonishing difference is simply that the Ordered\_Lists data structure is ordered, and it is trivial to access neighbouring cells. This is in contrast to the List\_2D structure, which is unordered and requires either a sort or a more sophisticated data structure to create the ability to access neighbouring cells in a similar fashion.

Conclusion

The three strongest approaches to solving the Wireworlds problem are the Unordered List, the Radix Trie, and the Headless 3-Tuples. These were all benchmarked on 100 transitions on the four Langton Wireworlds. The data has been already been presented, however it is reproduced here in full to allow comparisons between these approaches to be made.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Wireworld** | **Langton's 3x3** | **Langton's 5x5** | **Langton's 7x7** | **Langton's 11x11** |
| Unordered List | 0.190s | 0.494s | 0.952s | 2.304s |
| Radix Trie | 0.112s | 0.267s | 0.510s | 1.210s |
| Headless 3-Tuples | 0.152s | 0.433s | 0.870s | 2.178s |

Each of these solutions is clearly linear with respect to the size of the Wireworld as a whole. However, as noted in Problem 2 above, only the Headless 3-Tuple solution offers constant complexity with respect to the number of heads in the world. The radix trie solution, while faster than any other solution when tested against the Langton Wireworlds, deteriorates as the number of head cells increases. This could be a result of the cost of constructing the trie. There may be some other explanation for this behaviour that is not clear to the author[[2]](#footnote-3).

In any event, it is for this reason that the author prefers the headless 3-tuple over every other solution. While it is not as fast as the radix trie, it does not slow down as the proportion of different cell types changes throughout the simulation. In this sort of application, consistency of speed would be considered more important than raw runtime.

Extending Wireworld – Serial Binary Counter

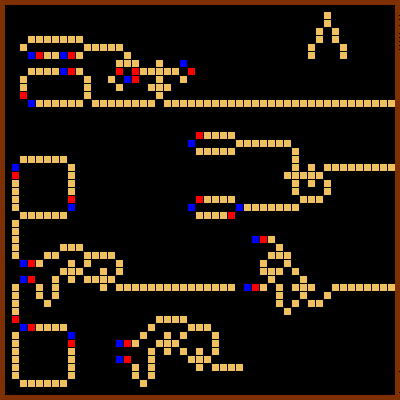
The extension problem was a completely different programming task to the first two problems.

The first stage was to get some basic logic gates working. Thanks to Google and websites such as Karl Sherer's Wireworld fanpage,(\*\*\*\*\*ref\*\*\*\*\*) it was fairly straightforward to obtain designs for logic gates that worked in 4-cycle time. However, the poor layout of that website and the absence of any useful documentation meant that a lot of work needed to be done to actually convert the designs into usable patterns. In particular, most of the gates that operated with 4-cycle clocks had to be very carefully configured to ensure that incoming electron heads were perfectly synchronised with those clocks.

Moreover, some of the designs on that website just do not work as advertised. In particular, the 4-cycle AND gates do not function correctly – at least, the author could not make them do so. But this error was not discovered until well into the design process; more on that later.

Using ordinary logic gates, the design of a binary counter is quite straightforward. Anyone familiar with binary counters will know that they can be constructed using AND gates and toggle (single input) flip flops. Not being familiar with binary counters, the author had to appeal to the University of Sydney (\*\*\*\*\*\*\*\*\*\*ref\*\*\*) to obtain that information.

Armed with this knowledge, the binary counter was slowly starting to form in the authors mind. A cascade of toggle flip flops and AND gates, wired correctly, should form a working binary counter. The toggle input of a flip flop n would have to be [the clock signal ^ ff0 ^ ff1 ^ .. ff(n-1)]. Fortunately, this could be expressed as a recursive function of sorts. Since the input of the n-1th flip flop was exactly the same, only without the output from itself, the solution found was the the input to a flip flop was simply *the* *conjunction of the previous flip flop's input with its own output*.

Illustration 1: AND gates that do not AND

The next step was to design a flipflop/AND gate 'cell'. The flip flop component was straightforward enough; the design was borrowed from Karl S's website (\*\*ref\*\*) and it worked perfectly. Designs for the AND gate were also borrowed from that website, with varying degrees of success.

There were some timing issues to resolve. If these cells were chained together naively, the very first clock signal could trigger a cascade of toggling down each flip flop. It was important that the correct clock signal be aligned with the outputs of the flip flop. For that reason, a slight delay was introduced by lengthening the approach of each flip flop's output to the AND gate immediately following it. This had the effect of 'holding back' a flip flop's output until the clock signal that had triggered that output had been blocked by the AND gate. This also had the effect of desynchronising the each output line, but finding a way to correct that issue was deferred until later.

As already mentioned, it was found that the 4-cycle AND gate on Karl's website did not function – or could not be made to function – as expected. In particular, if a steady stream of electrons was passed through the lower input wire, they would pass through the AND gate unimpeded. Oddly enough, if the stream was broken up, the AND gate would function properly. This error was discovered once the eight-bit structure had already been constructed by cascading the flip flop/AND cell. The structure was working perfectly when the electrons were 8, 12 and 16 bits apart, but once the cycle time was reduced to 4, the system fell apart.

At this point, the author decided to design an AND gate manually, using the NOT and ANDNOT gates found on Karl's website. There were synchronisation issues to deal with as both of those gates had 4-cycle clocks built into them. Once they were solved, however, the final design of the AND gate worked perfectly on 4 cycle timing. It is interesting to compare the working AND gate with the design on Karl's website – the difference is minute; only the location of one of the inverters is slightly different.

<< insert picture of and vs and here>>

The structure was reconstructed with the new, working AND gate, and it worked like a charm. The final step in the design process was to synchronise the outputs of each flip flop so that they all appeared in perfect lockstep on the output lines.

The output that the structure was generating was out of time, but it was systematically so. Each line was producing output that was about eight cells in advance of the line below it. This was undoubtedly because of the delay that was introduced in the input lines of the AND gates, to prevent cascades as discussed previously. To solve this issue, then, kinks were introduced to the output lines above each cell. This resulted in every output line remaining synchronised with the action happening in the flip flops immediately below them.

# This could be .. reflections

One strange observation that has been made concerns the decision to dispose of empty cells, in problem one. As reported above, reprogramming the transition\_world function to reject empty cells, rather than continuously transition them into new empty cells, resulted in an improvement to the overall runtime of the program of up to 25%.

However, when traceShow was being used to debug the radix trie data structure, the unmodified contents of 'world' were dumped to the screen, and the author noted that *there were are no empty cells to begin with*. So where did the 25% improvement come from[[3]](#footnote-4)? The mind boggles.

The author also notes that it is a *very* satisfying feeling when the data collected by measuring runtimes exactly matches the complexity yielded by a 'theoretical' analysis of the algorithm.

The construction of the binary counter within Wireworld was a completely different activity, and yet it is interesting to note that the same principles of recursive functions in Haskell could be used to achieve outcomes in Wireworld. Partially defining a cell in terms of the cell that came immediately before it greatly simplified the wiring task. It was a leap of logic that the author did not expect. It also yielded a very simple, repetitive structure. Dare I say, elegant? Looking at the binary counter, the repetitive elements are very easy to identify. It looks like a recursive Haskell function that has been 'unwrapped', or like one of those matryoshka dolls.

Illustration 2: Programming is like playing with dolls

References

<http://karlscherer.com/Wireworld.html>

<http://www.ee.usyd.edu.au/tutorials/digital_tutorial/part3/example1-6.htm>

<http://blog.exuberantanimal.com/home/41308/domains/blog.exuberantanimal.com/html/wp-content/uploads/2013/08/russian-dolls-opt.jpg>(picture of dolls)

There are no 'personal communications' to reference. The author avoids interacting with humans wherever possible.

Appendices

Appendix I The Wireworlds used for testing

Appendix II Measurements of program runtimes

Appendix III Source code of various implementations

# Appendix I

Wireworlds

A number of different Wireworld schemas were used to test these functions. The number of cells is the most important factor in differentiating them, so that information is summarised here.

Images of these Wireworlds have not been included as they may be found within the Wireworld zip package.

|  |  |
| --- | --- |
| **Wireworld** | **# of Cells** |
| Default | 655 |
| Langton's 3x3 | 4338 |
| Langton's 5x5 | 11805 |
| Langton's 7x7 | 22845 |
| Langton's 11x11 | 55965 |
| 7 Digit Display | 4649 |
|  |  |
|  |  |

# Appendix II

Runtimes

Benchmarking took place in one session on one computer without interruption, to minimise the influence of external variables such as differences in computer processor, memory, or available hard disk space.

For\_List Raw Data

Mark 1 – Naive approach

Tested on the default bitmap only

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test #** | **25 tr.** | **100 tr.** | **250 tr.** | **500 tr.** |
| 1 | 0.196 | 0.803 | 2.152 | 4.116 |
| 2 | 0.198 | 0.796 | 2.159 | 4.108 |
| 3 | 0.195 | 0.804 | 2.177 | 4.109 |
| 4 | 0.200 | 0.810 | 2.134 | 4.134 |
| 5 | 0.196 | 0.802 | 2.151 | 4.096 |
| 6 | 0.197 | 0.786 | 2.123 | 4.116 |
| 7 | 0.198 | 0.787 | 2.154 | 4.109 |
| 8 | 0.205 | 0.799 | 2.114 | 4.085 |
| 9 | 0.198 | 0.801 | 2.165 | 4.172 |
| 10 | 0.198 | 0.808 | 2.143 | 4.080 |
| **Mean** | **0.1981** | **0.7996** | **2.1472** | **4.1125** |
| Standard Dev. | 0.0028 | 0.0080 | 0.0192 | 0.0262 |

Mark 2 – No empties

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test #** | **25 tr.** | **100 tr.** | **250 tr.** | **500 tr.** |
| 1 | 0.174 | 0.629 | 1.560 | 3.097 |
| 2 | 0.168 | 0.636 | 1.570 | 3.097 |
| 3 | 0.174 | 0.639 | 1.554 | 3.107 |
| 4 | 0.175 | 0.628 | 1.565 | 3.107 |
| 5 | 0.173 | 0.631 | 1.563 | 3.104 |
| 6 | 0.175 | 0.635 | 1.569 | 3.100 |
| 7 | 0.174 | 0.634 | 1.558 | 3.103 |
| 8 | 0.175 | 0.626 | 1.574 | 3.095 |
| 9 | 0.167 | 0.630 | 1.563 | 3.089 |
| 10 | 0.169 | 0.636 | 1.565 | 3.098 |
| **Mean** | **0.1724** | **0.6324** | **1.5641** | **3.0997** |
| Standard Dev | 0.0031 | 0.0042 | 0.0059 | 0.0057 |

Mark 3 – Counting Neighbours

Default Bitmap

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test #** | **25 tr.** | **100 tr.** | **250 tr.** | **500 tr.** |
| 1 | 0.015 | 0.083 | 0.207 | 0.466 |
| 2 | 0.008 | 0.090 | 0.210 | 0.457 |
| 3 | 0.018 | 0.087 | 0.215 | 0.455 |
| 4 | 0.011 | 0.084 | 0.213 | 0.455 |
| 5 | 0.012 | 0.082 | 0.211 | 0.462 |
| 6 | 0.008 | 0.096 | 0.212 | 0.458 |
| 7 | 0.009 | 0.087 | 0.217 | 0.459 |
| 8 | 0.005 | 0.086 | 0.217 | 0.458 |
| 9 | 0.009 | 0.088 | 0.213 | 0.458 |
| 10 | 0.009 | 0.086 | 0.212 | 0.459 |
| **Mean** | **0.0104** | **0.0869** | **0.2127** | **0.4587** |
| Standard Dev | 0.0038 | 0.0040 | 0.0031 | 0.0033 |

Langton's 3x3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test #** | **25 tr.** | **100 tr.** | **250 tr.** | **500 tr.** |
| 1 | 0.040 | 0.189 | 0.650 | 1.824 |
| 2 | 0.040 | 0.192 | 0.648 | 1.839 |
| 3 | 0.032 | 0.186 | 0.652 | 1.823 |
| 4 | 0.042 | 0.187 | 0.651 | 1.836 |
| 5 | 0.035 | 0.191 | 0.653 | 1.830 |
| 6 | 0.039 | 0.194 | 0.650 | 1.826 |
| 7 | 0.044 | 0.187 | 0.643 | 1.841 |
| 8 | 0.044 | 0.193 | 0.659 | 1.836 |
| 9 | 0.044 | 0.186 | 0.651 | 1.827 |
| 10 | 0.039 | 0.192 | 0.649 | 1.843 |
| **Mean** | **0.0399** | **0.1897** | **0.6506** | **1.8325** |
| Standard Dev | 0.0040 | 0.0031 | 0.0040 | 0.0074 |

100 transitions on Langtons of different sizes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test #** | **L 3x3** | **L 5x5** | **L 7x7** | **L 11x11** |
| 1 | 0.189 | 0.495 | 0.946 | 2.290 |
| 2 | 0.192 | 0.496 | 0.950 | 2.308 |
| 3 | 0.186 | 0.495 | 0.954 | 2.302 |
| 4 | 0.187 | 0.490 | 0.958 | 2.316 |
| 5 | 0.191 | 0.494 | 0.954 | 2.307 |
| 6 | 0.194 | 0.493 | 0.955 | 2.305 |
| 7 | 0.187 | 0.498 | 0.956 | 2.303 |
| 8 | 0.193 | 0.490 | 0.947 | 2.311 |
| 9 | 0.186 | 0.491 | 0.948 | 2.301 |
| 10 | 0.192 | 0.494 | 0.952 | 2.300 |
| **Mean** | **0.1897** | **0.4936** | **0.9520** | **2.3043** |
| Standard Dev | 0.0031 | 0.0026 | 0.0041 | 0.0070 |

Seven Digit Display

This particular Wireworld was selected because it generates a large number of heads. As the number of transitions increases, so too does the proportion of head cells to total cells in the world.

Testing on 250 transitions was aborted when it became apparent that it would take a minute to collect each measurement.

|  |  |  |  |
| --- | --- | --- | --- |
| **Test #** | **25 tr.** | **100 tr.** | **250 tr.** |
| 1 | 0.052 | 1.450 | 35.421 |
| 2 | 0.061 | 1.448 | 35.318 |
| 3 | 0.056 | 1.452 | - |
| 4 | 0.059 | 1.440 | - |
| 5 | 0.057 | 1.453 | - |
| 6 | 0.059 | 1.452 | - |
| 7 | 0.053 | 1.445 | - |
| 8 | 0.049 | 1.452 | - |
| 9 | 0.063 | 1.459 | - |
| 10 | 0.055 | 1.456 | - |
| **Mean** | **0.0564** | **1.4507** | **35.3695** |
| Standard Dev | 0.0043 | 0.0054 | 0.07283 |

Mark 4 – A Trie by any other name

100 transition Langton benchmarking

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test #** | **L 3x3** | **L 5x5** | **L 7x7** | **L 11x11** |
| 1 | 0.109 | 0.262 | 0.51 | 1.199 |
| 2 | 0.113 | 0.267 | 0.51 | 1.204 |
| 3 | 0.117 | 0.266 | 0.514 | 1.219 |
| 4 | 0.107 | 0.271 | 0.51 | 1.209 |
| 5 | 0.116 | 0.264 | 0.511 | 1.222 |
| 6 | 0.109 | 0.27 | 0.508 | 1.211 |
| 7 | 0.116 | 0.264 | 0.508 | 1.203 |
| 8 | 0.11 | 0.273 | 0.511 | 1.203 |
| 9 | 0.108 | 0.265 | 0.507 | 1.225 |
| 10 | 0.114 | 0.271 | 0.511 | 1.206 |
| **Mean** | **0.1119** | **0.2673** | **0.51** | **1.2101** |
| Standard Dev | 0.0037 | 0.0037 | 0.002 | 0.0090 |

Seven Digit Display

A measurement of performance vs number of heads

|  |  |  |  |
| --- | --- | --- | --- |
| **Test #** | **25 tr.** | **100 tr.** | **250 tr.** |
| 1 | 0.025 | 0.269 | 2.451 |
| 2 | 0.025 | 0.271 | 2.454 |
| 3 | 0.029 | 0.269 | 2.423 |
| 4 | 0.029 | 0.266 | 2.408 |
| 5 | 0.031 | 0.273 | 2.416 |
| 6 | 0.032 | 0.268 | 2.396 |
| 7 | 0.03 | 0.272 | 2.391 |
| 8 | 0.033 | 0.274 | 2.401 |
| 9 | 0.026 | 0.265 | 2.399 |
| 10 | 0.032 | 0.269 | 2.395 |
| **Mean** | **0.0292** | **0.2696** | **2.4134** |
| Standard Dev | 0.0030 | 0.0029 | 0.0228 |

Mark 5 – Really Bad Optimisations

Benchmarking radix tries of different sizes

Langton's 11x11 on 1000 transitions

|  |  |  |  |
| --- | --- | --- | --- |
| **Test #** | **Radix 4** | **Radix 2** | **Radix 8** |
| 1 | 12.351 | 16.211 | 53.198 |
| 2 | 12.434 | 16.258 | - |
| 3 | 12.349 | 16.194 | - |
| 4 | 12.558 | 16.263 | - |
| **Mean** | **12.423** | **16.2315** | **53.198** |
| Standard Dev | 0.0983 | 0.0343 | - |

Problem 2 Datasets

Mark I – Naive Solution

Benchmarking on Langton's 3x3 → 11x11

100 transitions

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test #** | **L 3x3** | **L 5x5** | **L 7x7** | **L 11x11** |
| 1 | 0.641 | 3.01 | 8.738 | 36.421 |
| 2 | 0.632 | 3.064 | 8.746 | - |
| 3 | 0.648 | 3.079 | 8.877 | - |
| 4 | 0.654 | 3.06 | 8.767 | - |
| 5 | 0.654 | 3.091 | - | - |
| 6 | 0.641 | 3.081 | - | - |
| 7 | 0.638 | 3.075 | - | - |
| 8 | 0.664 | 3.067 | - | - |
| 9 | 0.643 | 3.057 | - | - |
| 10 | 0.647 | 3.092 | **-** | **-** |
| **Mean** | **0.6462** | **3.0676** | **8.782** | **36.421** |
| Standard Dev | 0.0093 | 0.0236 | 0.0645 | - |

Mark 2 – Nearly Headless Lists

Benchmarking on Langton's 3x3 → 11x11

100 transitions

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test #** | **L 3x3** | **L 5x5** | **L 7x7** | **L 11x11** |
| 1 | 0.157 | 0.429 | 0.863 | 2.174 |
| 2 | 0.155 | 0.427 | 0.866 | 2.177 |
| 3 | 0.148 | 0.422 | 0.866 | 2.175 |
| 4 | 0.147 | 0.433 | 0.858 | 2.181 |
| 5 | 0.153 | 0.440 | 0.870 | 2.175 |
| 6 | 0.156 | 0.431 | 0.877 | 2.173 |
| 7 | 0.152 | 0.432 | 0.880 | 2.175 |
| 8 | 0.150 | 0.439 | 0.867 | 2.189 |
| 9 | 0.156 | 0.439 | 0.874 | 2.170 |
| 10 | 0.149 | 0.437 | 0.874 | 2.190 |
| **Mean** | **0.1523** | **0.4329** | **0.8695** | **2.1779** |
| Standard Dev | 0.0037 | 0.0059 | 0.0068 | 0.0067 |

Seven Digit Display

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test #** | **25 tr.** | **100 tr.** | **250 tr.** | **50 tr.** |
| 1 | 0.044 | 0.151 | 0.367 | 0.712 |
| 2 | 0.051 | 0.154 | 0.365 | 0.728 |
| 3 | 0.046 | 0.157 | 0.368 | 0.725 |
| 4 | 0.041 | 0.152 | 0.362 | 0.716 |
| 5 | 0.050 | 0.150 | 0.365 | 0.728 |
| 6 | 0.042 | 0.159 | 0.366 | 0.721 |
| 7 | 0.048 | 0.155 | 0.362 | 0.719 |
| 8 | 0.040 | 0.159 | 0.361 | 0.716 |
| 9 | 0.046 | 0.153 | 0.371 | 0.722 |
| 10 | 0.051 | 0.161 | 0.367 | 0.726 |
| **Mean** | **0.0459** | **0.1551** | **0.3654** | **0.7213** |
| Standard Dev | 0.0041 | 0.0038 | 0.0031 | 0.0055 |

Source Codes

For\_List2D

Mark 1:

-- First attempt - O(n) as it goes through the list from start to finish,

-- and O(n) again as it calls functions that are O(n)

-- so this function is O(n squared)

-- yuck!

transition\_world :: List\_2D Cell -> List\_2D Cell

transition\_world world = transition\_world\_recurser world world

-- this function exists so it can 'hang on' to the full size world

transition\_world\_recurser :: List\_2D Cell -> List\_2D Cell -> List\_2D Cell

transition\_world\_recurser world full\_world = case world of

(e, (x, y)) : es -> (new\_cell\_state, (x, y)) : transition\_world\_recurser es full\_world

where new\_cell\_state = compute\_new\_cell\_state (e, (x, y)) full\_world

[] -> []

-- this one was [redacted] easy

compute\_new\_cell\_state :: Element\_w\_Coord Cell -> List\_2D Cell -> Cell

compute\_new\_cell\_state (e, (x, y)) world = case e of

Head -> Tail

Tail -> Conductor

Empty -> Empty

Conductor

| (num\_heads == 1 || num\_heads == 2) -> Head

| otherwise -> Conductor

where num\_heads = element\_occurrence Head (local\_elements (x, y) world)

Mark 2:

-- Second attempt – still O(n squared) mate

-- yuck!

transition\_world :: List\_2D Cell -> List\_2D Cell

transition\_world world = transition\_world\_recurser world world

-- this function exists so it can 'hang on' to the full size world

transition\_world\_recurser :: List\_2D Cell -> List\_2D Cell -> List\_2D Cell

transition\_world\_recurser world full\_world = case world of

(e, (x, y)) : es

**| (new\_cell\_state == Empty) -> transition\_world\_recurser es full\_world**

| otherwise -> (new\_cell\_state, (x, y)) : transition\_world\_recurser es full\_world

where new\_cell\_state = compute\_new\_cell\_state (e, (x, y)) full\_world

[] -> []

-- this one was [redacted] easy

compute\_new\_cell\_state :: Element\_w\_Coord Cell -> List\_2D Cell -> Cell

compute\_new\_cell\_state (e, (x, y)) world = case e of

Head -> Tail

Tail -> Conductor

Empty -> Empty

Conductor

| (num\_heads == 1 || num\_heads == 2) -> Head

| otherwise -> Conductor

where num\_heads = element\_occurrence Head (local\_elements (x, y) world)

Mark 3 – Counting Neighbours

transition\_world :: List\_2D Cell -> List\_2D Cell

transition\_world world = transition\_world\_recurser world (create\_neighbours world [])

transition\_world\_recurser :: List\_2D Cell -> List\_2D Nat -> List\_2D Cell

transition\_world\_recurser world neighbours = case world of

(e, (x, y)) : es

| (new\_cell\_state == Empty) -> transition\_world\_recurser es neighbours

| otherwise -> (new\_cell\_state, (x, y)) : transition\_world\_recurser es neighbours

where new\_cell\_state = compute\_new\_cell\_state (e, (x, y)) neighbours

[] -> []

compute\_new\_cell\_state :: Element\_w\_Coord Cell -> List\_2D Nat -> Cell

compute\_new\_cell\_state (e, (x, y)) neighbours = case e of

Head -> Tail

Tail -> Conductor

Empty -> Empty

Conductor -> case num\_heads of

Just 1 -> Head

Just 2 -> Head

\_ -> Conductor

where num\_heads = read\_element (x, y) neighbours

create\_neighbours :: List\_2D Cell -> List\_2D Nat -> List\_2D Nat

create\_neighbours world working = case world of

[] -> working

(e, (x, y)) : es -> case e of

Head -> create\_neighbours es (insert\_neighbour (x, y) working)

\_ -> create\_neighbours es working

insert\_neighbour :: Coord -> List\_2D Nat -> List\_2D Nat

insert\_neighbour (x, y) neighbours = insert\_recurser neighbour\_set neighbours

where neighbour\_set = [(a, b) | a <- [x-1..x+1], b <- [y-1..y+1], (a /= x || b /= y)]

insert\_recurser :: [Coord] -> List\_2D Nat -> List\_2D Nat

insert\_recurser coords neighbours = case coords of

[] -> neighbours

c : cs -> insert\_recurser cs (insert\_into' c neighbours)

insert\_into' :: Coord -> List\_2D Nat -> List\_2D Nat

insert\_into' (x, y) neighbours = case neighbours of

(e, (x', y')) : ls

| (x == x' && y == y') -> (e + 1, (x, y)) : ls

| otherwise -> (e, (x', y')) : insert\_into' (x, y) ls

[] -> [(1, (x, y))]

Mark 5 – Inexplicably Bad Optimisations

This rewrite of the main transition\_world function was supposed to decrease the running time of the function. For reasons best known to Haskell, it *quadrupled* the runtime instead.

transition\_world :: List\_2D Cell -> List\_2D Cell

transition\_world world = transition\_conductors working (create\_neighbours heads) conductors

where (working, heads, conductors) = part\_process\_world world ([], [], [])

-- this function accepts the working list of the world, the neighbours radix trie,

-- and the list of conductor coordinates. It returns the complete new world in its entirety.

transition\_conductors :: List\_2D Cell -> FirstRadixNode (Maybe (FirstRadixNode (Maybe Nat))) -> [Coord] -> List\_2D Cell

transition\_conductors new\_world neighbours conductors = case conductors of

[] -> new\_world

c: cs -> case count\_neighbour c neighbours of

1 -> transition\_conductors ((Head , c): new\_world) neighbours cs

2 -> transition\_conductors ((Head , c): new\_world) neighbours cs

\_ -> transition\_conductors ((Conductor, c): new\_world) neighbours cs

-- this function will go through the contents of the World list once only, and produce a troople

-- of three lists - the first list is the working New\_World, where all Tails -> Conductor and Head -> Tail

-- are included - and Empties dumped. The second list is a list of coordinates of all the Heads (that have been

-- transformed into tails). This list will be used to generate the neighbours Radix Trie. The last list

-- contains coords of all the Conductors, which have not been processed yet. Once the neighbours radix is created,

-- this last list will be processed to produce new Head

--

-- lawl

part\_process\_world :: List\_2D Cell -> (List\_2D Cell, [Coord], [Coord]) -> (List\_2D Cell, [Coord], [Coord])

part\_process\_world world (new\_world, heads, conductors) = case world of

[] -> (new\_world, heads, conductors)

(e, (x, y)): es -> case e of

Empty -> part\_process\_world es ( new\_world, heads, conductors)

Tail -> part\_process\_world es ((Conductor, (x, y)) : new\_world, heads, conductors)

Head -> part\_process\_world es (( Tail, (x, y)) : new\_world, (x, y): heads, conductors)

Conductor -> part\_process\_world es ( new\_world, heads, (x, y): conductors)

Problem 2 Sources

Mark 1 – Naive Solution

transition\_world :: Ordered\_Lists\_2D Cell -> Ordered\_Lists\_2D Cell

transition\_world world = transition\_world\_recurser (return\_line\_triplets world) []

where

transition\_world\_recurser :: [(Sparse\_Line Cell, Sparse\_Line Cell, Sparse\_Line Cell)] -> Ordered\_Lists\_2D Cell -> Ordered\_Lists\_2D Cell

transition\_world\_recurser triplets new\_world = case triplets of

[] -> new\_world

x: xs -> transition\_world\_recurser xs (transition\_line x new\_world)

return\_line\_triplets :: Ordered\_Lists\_2D Cell -> [(Sparse\_Line Cell, Sparse\_Line Cell, Sparse\_Line Cell)]

return\_line\_triplets world = case world of

[] -> []

[x] -> [(Sparse\_Line ((y\_pos x) - 1) [], x, Sparse\_Line ((y\_pos x) + 1) [])]

x: y: rest -> (Sparse\_Line ((y\_pos x) - 1) [], x, y): return\_line\_triplet\_recurser (x: y: rest)

where return\_line\_triplet\_recurser :: Ordered\_Lists\_2D Cell -> [(Sparse\_Line Cell, Sparse\_Line Cell, Sparse\_Line Cell)]

return\_line\_triplet\_recurser smaller\_world = case smaller\_world of

[] -> error "fuckin error m8"

[\_] -> error "get stuffed"

[prev, x] -> [(prev, x, Sparse\_Line ((y\_pos x) + 1) [])]

prev: x: next: rest -> (prev, x, next) : return\_line\_triplet\_recurser (x: next: rest)

-- take a list of old entries, transition them, and return a new list of entries

-- I don't even know why I need the y coordinate for this

transition\_line :: (Sparse\_Line Cell, Sparse\_Line Cell, Sparse\_Line Cell) -> Ordered\_Lists\_2D Cell -> Ordered\_Lists\_2D Cell

transition\_line (prev, a, next) new\_world = case e of

[] -> new\_world

\_ -> (Sparse\_Line y (transition\_line\_recurser e l)): new\_world

where e = entries a

y = y\_pos a

l = (prev, a, next)

transition\_line\_recurser :: Placed\_Elements Cell -> (Sparse\_Line Cell, Sparse\_Line Cell, Sparse\_Line Cell) -> Placed\_Elements Cell

transition\_line\_recurser old\_elements l' = case old\_elements of

[] -> []

x: xs -> Placed\_Element (x\_pos x) (transition\_cell x l'): transition\_line\_recurser xs l'

transition\_cell :: Placed\_Element Cell -> (Sparse\_Line Cell, Sparse\_Line Cell, Sparse\_Line Cell) -> Cell

transition\_cell cell l = case entry cell of

Empty -> Empty

Head -> Tail

Tail -> Conductor

Conductor -> case count\_heads cell l of -- ha, ha!

1 -> Head

2 -> Head

\_ -> Conductor

count\_heads :: Placed\_Element Cell -> (Sparse\_Line Cell, Sparse\_Line Cell, Sparse\_Line Cell) -> Int

count\_heads cell (a, b, c) = (count\_heads\_per\_line (entries a) target) + (count\_heads\_per\_line (entries b) target) + (count\_heads\_per\_line (entries c) target)

where target = (x\_pos cell) + 1

count\_heads\_per\_line :: Placed\_Elements Cell -> X\_Coord -> Int

count\_heads\_per\_line line target = case line of

[] -> 0

x: xs

| x\_pos x > target -> 0

| x\_pos x < (target - 2) -> count\_heads\_per\_line xs target

| entry (x) == Head -> 1 + count\_heads\_per\_line xs target

| otherwise -> count\_heads\_per\_line xs target

1. And the author's ego [↑](#footnote-ref-2)
2. The Gods of Haskell move in mysterious ways. [↑](#footnote-ref-3)
3. See note [2]. [↑](#footnote-ref-4)