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].

1 And the author's ego

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.

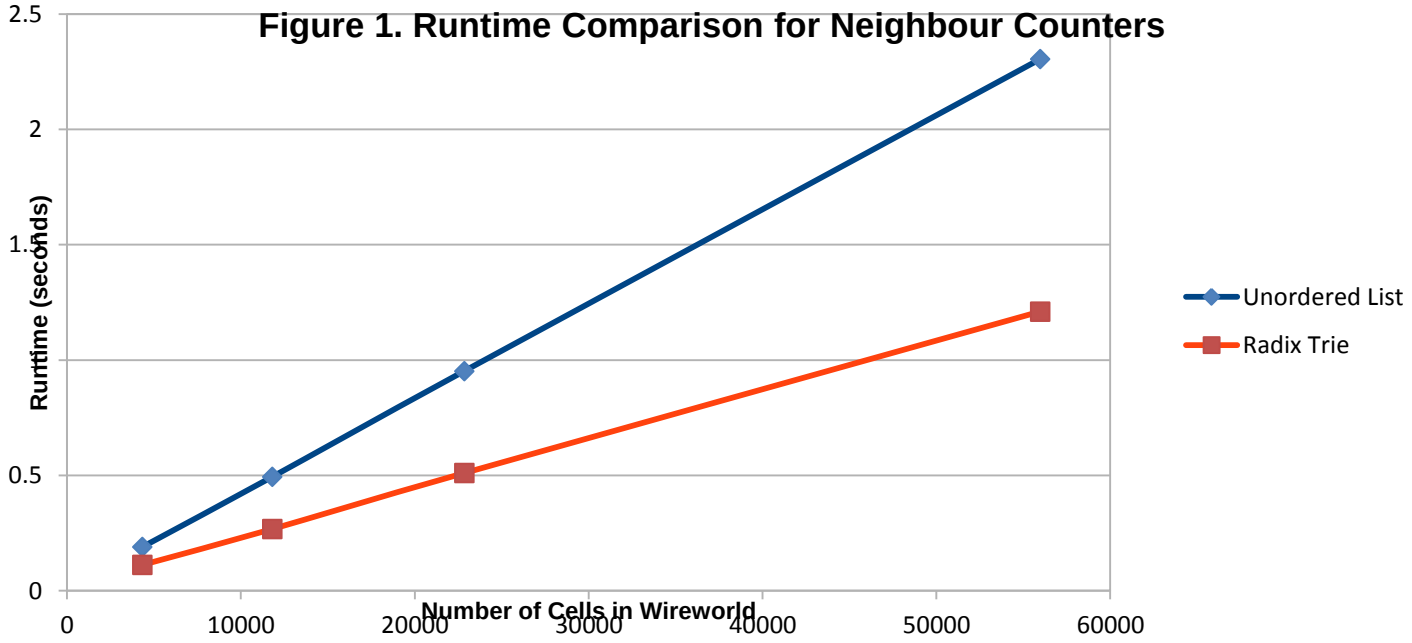
This had the effect of reducing the algorithmic complexity to O(n) with respect to the size of the world, as the 'world' was only being traversed twice. Of course, when talking about complexity, O(2n) == O(n).

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.

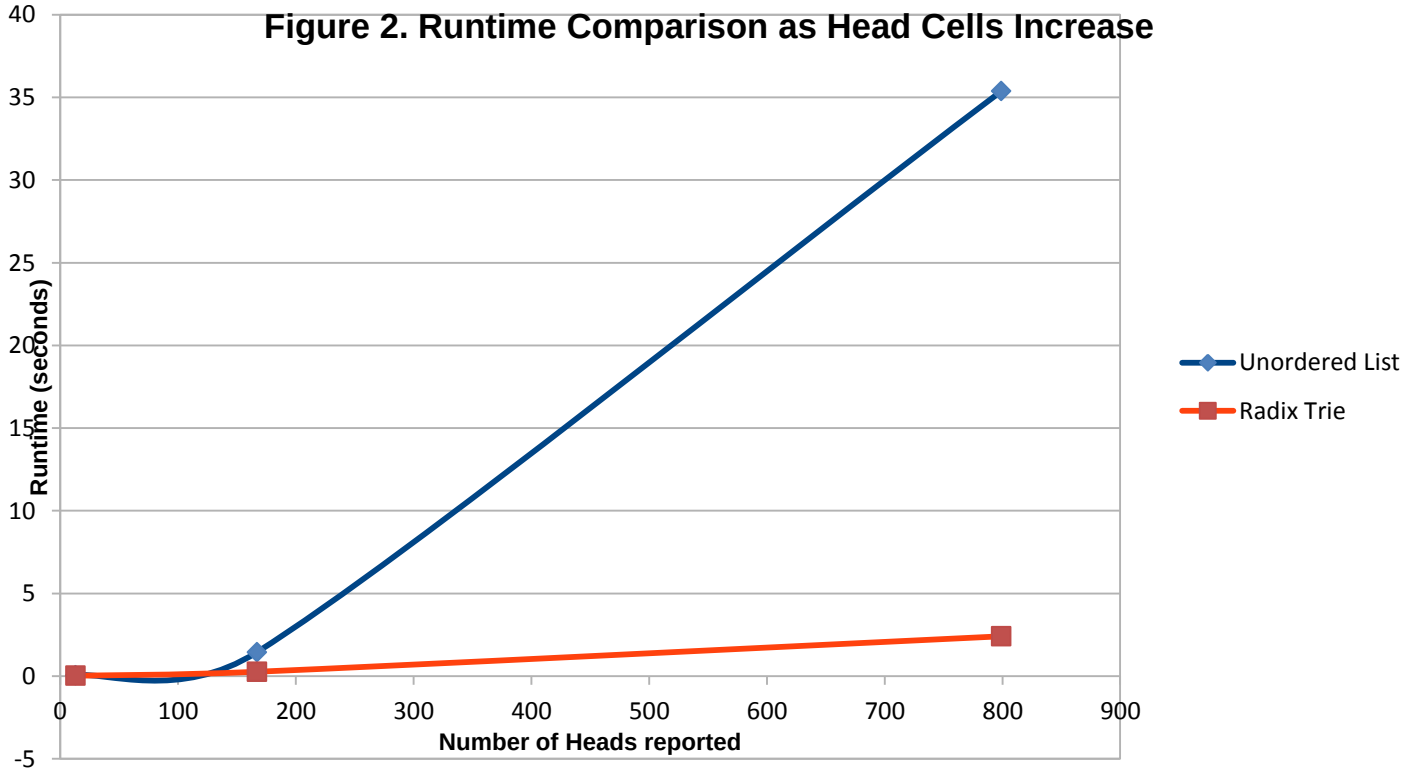
So the final data structure is a FirstRadixNode that returns a radix trie that returns a FirstRadixNode that returns a radix trie that returns an integer. O(1) time indeed.

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.

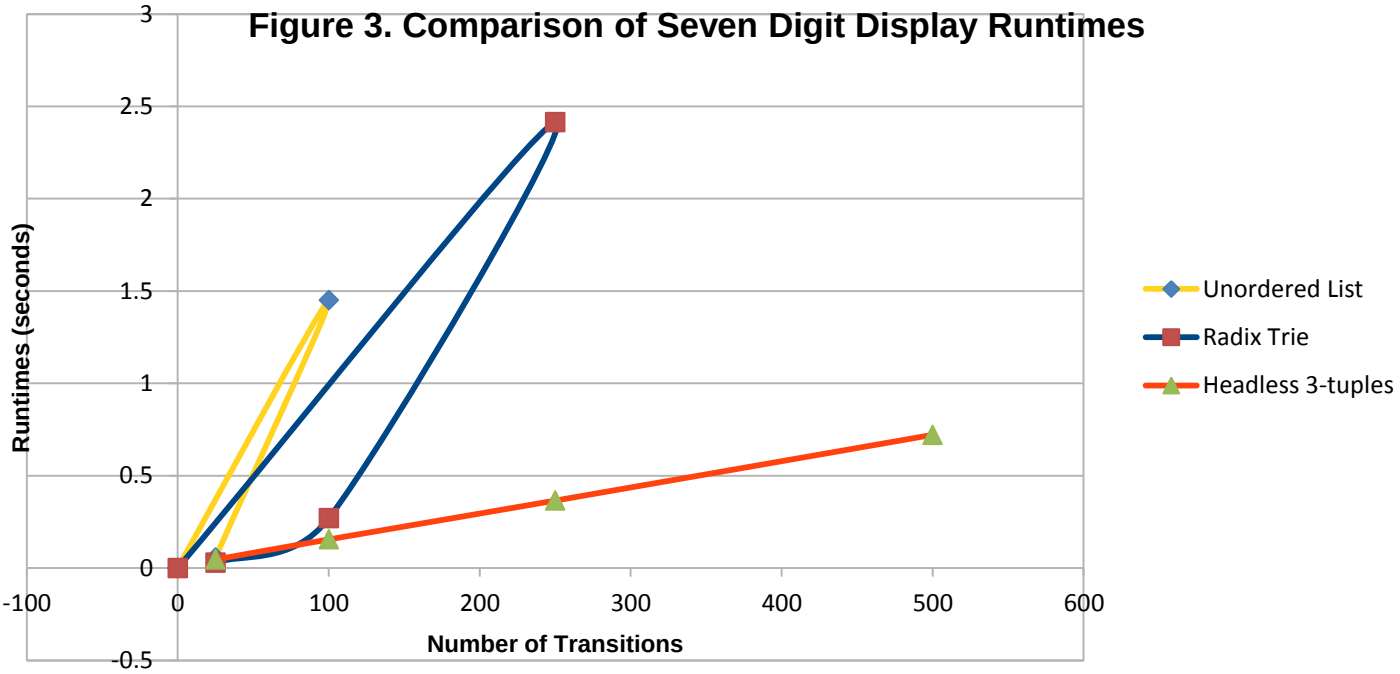
As per the previous solution, this was benchmarked against the four Langton Wireworlds on 100 transitions.

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.

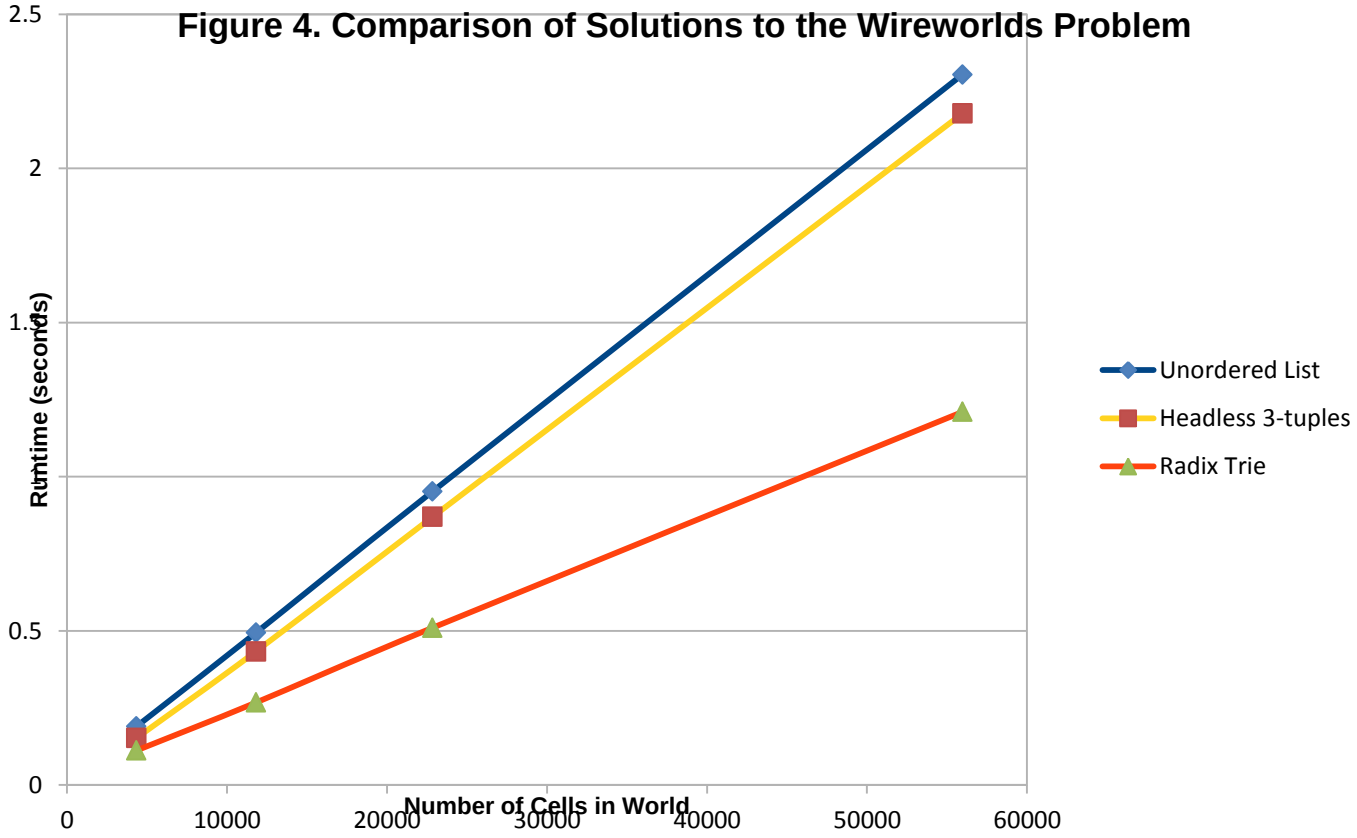


The explanation for this astonishing difference is simply that the Ordered_Lists data structure is ordered, and it is trivial to access neighbouring cells. This is in contrast to the List_2D structure, which is unordered and requires either a sort or a more sophisticated data structure to create the 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].

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

2 The Gods of Haskell move in mysterious ways.

cell types changes throughout the simulation. In this sort of application, consistency of speed would be considered more important than raw runtime.

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]? 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.

3 See note [2].

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
```