Assignment_1/Sources/Transitions/For_List_2D.hs
Geyang Huang

```haskell
-- Original:
-- Uwe R. Zimmer
-- Australia 2012
--
-- Modified by:
-- Huang Geyang
-- 16 March 2014
--

module Transitions.For_List_2D (
    transition_world -- :: List_2D Cell -> List_2D Cell
) where

import Data.Cell (Cell (Head, Tail, Conductor, Empty))
import Data.Coordinates
import Data.List_2D


transition_world :: List_2D Cell -> List_2D Cell
transition_world world = transition world world  --copy another world
    where   transition :: List_2D Cell -> List_2D Cell -> List_2D Cell
            transition world world_copy = case world of
                (Conductor, (x, y)) : xs -> case element_occurrence Head (local_elements (x, y
                    1 -> (Head, (x, y)) : transition xs world_copy
                    2 -> (Head, (x, y)) : transition xs world_copy
                    _ -> (Conductor, (x, y)) : transition xs world_copy
                (Head, (x, y)) : xs -> (Tail, (x, y)) : transition xs world_copy
                (Tail, (x, y)) : xs -> (Conductor, (x, y)) : transition xs world_copy
                (Empty, (x, y)) : xs -> (Empty, (x, y)) : transition xs world_copy
                [] -> []
```

Assignment_1/Sources/Transitions/For_Ordered_Lists_2D.hs
Geyang Huang

```
-- Original:
-- Uwe R. Zimmer
-- Australia 2012
--
-- Modified by:
-- Huang Geyang
-- u5421856
-- 17 March 2014
--
-- Updated by Huang Geyang @ 3 April 2014:
-- Simplify the local_space to just be the local_line function (by excluding the empty case)
-- Modify the given local_elements function (reduce checking steps)
--
-- I have the confidence in the efficiency of my program.
-- For example, it can run the "Playfield.bmp" up to 1200fps!
--                                       -- Huang Geyang

module Transitions.For_Ordered_Lists_2D (
    transition_world -- :: Ordered_Lists_2D Cell -> Ordered_Lists_2D Cell
) where

import Data.Cell (Cell (Head, Tail, Conductor, Empty))
import Data.Coordinates
import Data.Ordered_Lists_2D


transition_world :: Ordered_Lists_2D Cell -> Ordered_Lists_2D Cell
transition_world world = transition_line world world

    where   transition_line :: Ordered_Lists_2D Cell -> Ordered_Lists_2D Cell -> Ordered_Lists
            transition_line world world_copy = case world of
                [] -> []
                line : lines -> Sparse_Line {y_pos = y_pos line, entries = transition (entries

                    where
                        local_space :: Y_Coord -> Ordered_Lists_2D Cell -> Ordered_Lists_2D Ce
                        local_space y world_copy = (local_lines y world_copy)

                        transition :: Placed_Elements Cell -> Y_Coord -> Ordered_Lists_2D Cell
                        transition eline y space = case eline of
                            cell : cells -> transition_cell cell y space : transition cells y
                            [] -> []

                            where
                                transition_cell :: Placed_Element Cell -> Y_Coord -> Ordered_I
                                transition_cell cell y space = case entry cell of
                                    Conductor -> case element_occurrence Head (new_local_eleme
    1 -> (cell {entry = Head})
    2 -> (cell {entry = Head})
    _ -> (cell {entry = Conductor})
Head -> (cell {entry = Tail})
Tail -> (cell {entry = Conductor})
Empty -> (cell {entry = Empty})

-- Modified given local_elements function by Huang Geyang
where
    new_local_elements :: Coord -> Ordered_Lists_2D e -> O
    new_local_elements (x, y) space = case space of
        l:ls -> new_neighbours_in_line l: (new_local_eleme
        []   -> []

        where
            new_neighbours_in_line :: Sparse_Line e -> Spa
            new_neighbours_in_line line = line {entries =

                where
                    new_neighbours_in_entries :: Placed_El
                    new_neighbours_in_entries list = case
                        c: cs
                            | x < (x_pos c) - 1      -> []
                            | abs (x - x_pos c) <= 1 -> c:
                            | otherwise              ->
                        [] -> []
```

REPORT FOR WIRING UP WIREWORLD

Huang Geyang u5421856

Background

Wireworld is an interesting cellular automaton. When I first read about it, the ways how it works reminds me of the electronic logic elements. My first programming task is to implement two kinds of transition methods and to make the Wireworld move. I design my programs based on two principles. The first one is that my programs must be able to work orderly and give users the correct output - they must be accurate. The second one is my programs must be efficient. On top of this two I will try to make my programs easy to understand by others. Accuracy is the fundamental requirement for the programs, however the efficiency is also another important factor to be considered for this task. This is because the transition method will be called each time when the frame is updated, which is probably more than ten times per second. Moreover, since the input and output files are in bmp format, my program will most probably need to traverse the every pixel on the picture, so that the computational complexity is based on O(n^2), which is quite scary. Besides, I do not want to over complicated my programs either as the basic functions have been provided, if I create too many my own functions there will be huge costs in debugging. Therefore I need to balance minimising computational complexity with conceptional complexity for my programs.

Constructing For_List_2D.hs

Wireworld can be saved in form of **List_2D** by save each pixel's status with its x, y coordinates as a whole element into a list, with no particular orders. I started on For_List_2D.hs because it is actually simpler than the other program and it can help me understand how does the Wireworld work better. My algorithm is very simple, extract one element (consisting of a cell data with its coordinates) from the **List_2D** at a time, and change its status by the rules given. If it is an **Conductor**, the program will find the surrounding 8 cells using given function (**local_elements**), and count how many **Head**s there using **element_occurrence**, if there are 1 or 2, the cell will be changed to **Head**, otherwise no change on it. Other types of cell will change as following: **Head** to **Tail**, **Tail** to **Conductor** and **Empty** to **Empty**. I directly applied the algorithm above with recursion for the first attempt, but it did not work well. I realised that it is because the variable **world** has changed during the recursion process. Therefore I copied the original **world** to

world_copy at the first then passed down **world_copy** during recursion. It then worked well at the second time.

Constructing For_Ordered_Lists_2D.hs

Ordered_Lists_2D is a more efficient data structure to store the Wireworld compared to **List_2D**. In **Ordered_Lists_2D**, pixels are converted into **Placed_Element** consisting of their x coordinate and their cell states first, then for pixels with the same y coordinate they form a list called **Placed_Elements** orderly. The list forms a new data type called **Sparse_Line** with its y coordinate. Hence the **Ordered_Lists_2D** is just a ordered list of **Sparse_Line**. Since **Ordered_Lists_2D** preserves the order of original Wireworld in it, the program should be specially designed to take advantage of its property and be more efficient. My algorithm is modified from the above one. Instead of extracting one element at a time, my program extract one **Sparce_Line** from the **Ordered_Lists_2D** at a time. Since all **Placed_Element**s in the same **Sparce_Line** actually share neighbours from the same space(two neighbour **Sparce_Line** plus its own **Sparce_Line**), it is more efficient if we extract the local space as well at the time the **Sparce_Line** is extracted. This will highly reduce searching space for the counting **Head**s process in **transition_cell**(a sub function to update the cell states). Therefore I implement a new function, **local_space**(consisting of given functions **combine_Ordered_Lists_2D** and **local_lines**), to find the neighbour space for each **Sparce_Line**. I think this step is the key to make the difference in efficiency between For_List_2D.hs and For_Ordered_Lists_2D.hs, as avoid repeating calculation is crucial in optimising algorithm. The rest of the program is to split the list into single elements and update one by one.

At the latest update(after I write the report), I modify the **local_space** to just be **local_lines**.

Test and Analysis

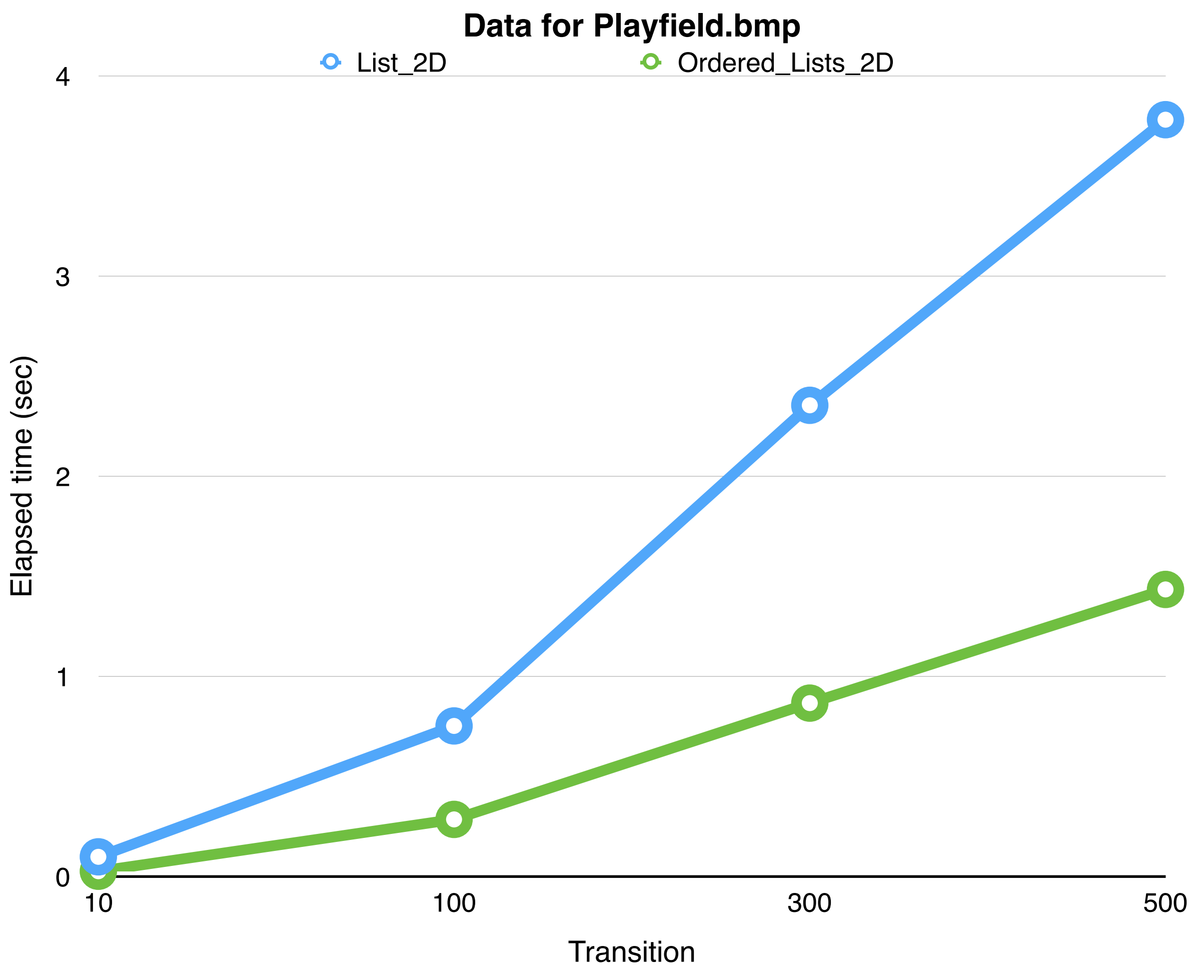
(The test was done before I did the last update, after updating, the efficiency of Ordered_Lists_2D is about **4 times** of the original one.)

To test the two programs above, I will run the same set of Wireworlds under different model (List_2D and Ordered_Lists_2D) on the same machine for multiple times. The following is a table of the data generated from testing.

Transition	Cell	Active Heads	Elapsed time (sec) for List_2D	Elapsed time (sec) for Ordered_Lists _2D	Elapsed time (sec) for the latest updated Ordered_Lists _2D
10	655	2	0.098	0.026	
100	655	24	0.752	0.285	
300	655	6	2.354	0.866	
500	655	16	3.781	1.434	0.380
100	12	1	0.002	0.002	
1000	12	1	0.008	0.018	
5000	12	1	0.028	0.075	
10000	12	1	0.068	0.150	
15000	12	1	0.090	0.224	0.122
1	3515	19	0.212	0.018	
2	3515	44	0.474	0.031	
3	3515	52	0.608	0.054	
4	3515	12	0.873	0.087	
5	3515	20	1.055	0.093	
20	3515	17	4.454	0.434	0.139
10	55965	3	Langton’s Ant 11x11 for testing		4.417

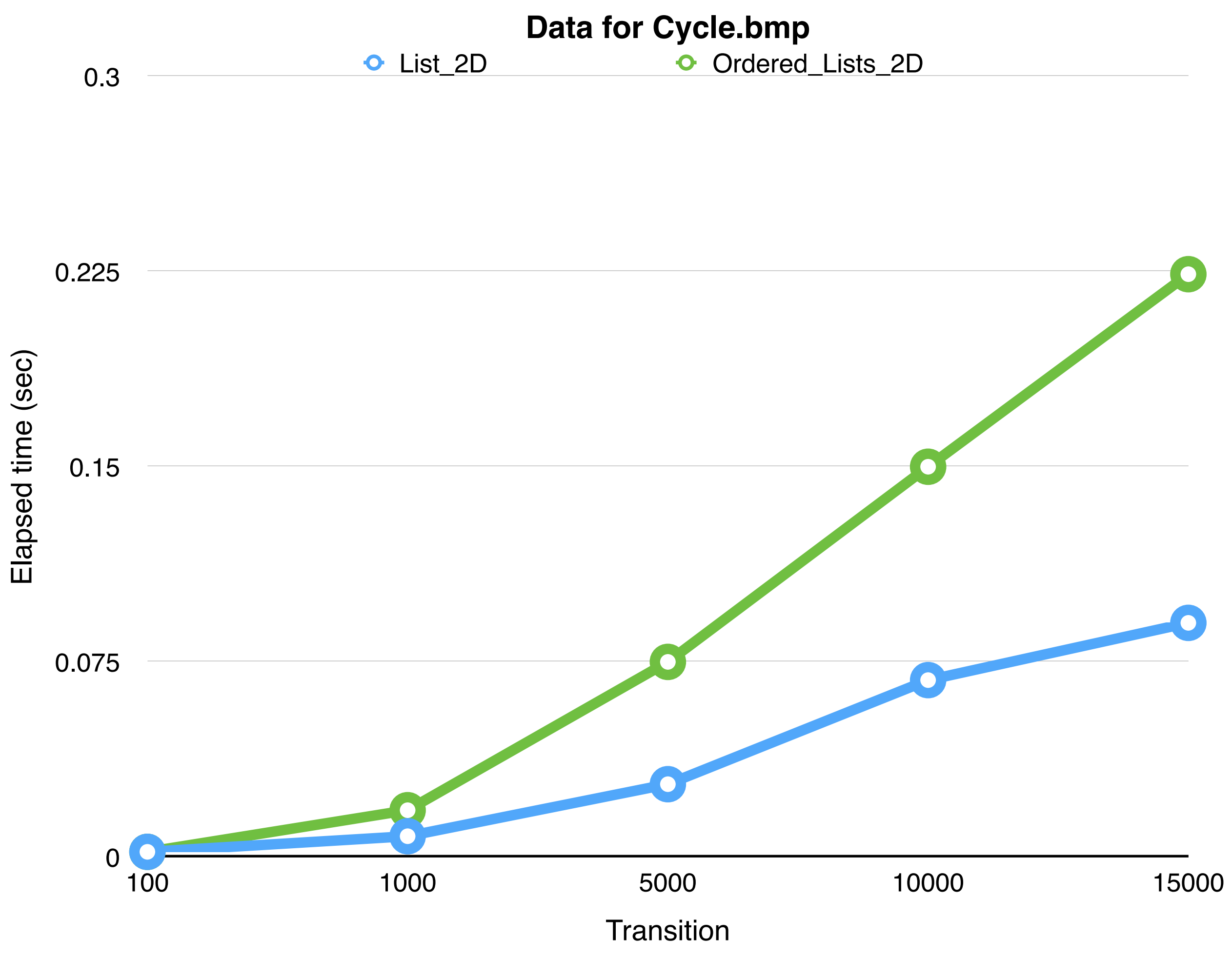
The results here are very interesting. There are three different Wireworlds corresponding to three different cell numbers.

The first one (655 cells) is the default Wireworld (Playfield.bmp), which is an average case for these two programs, based on my opinion.



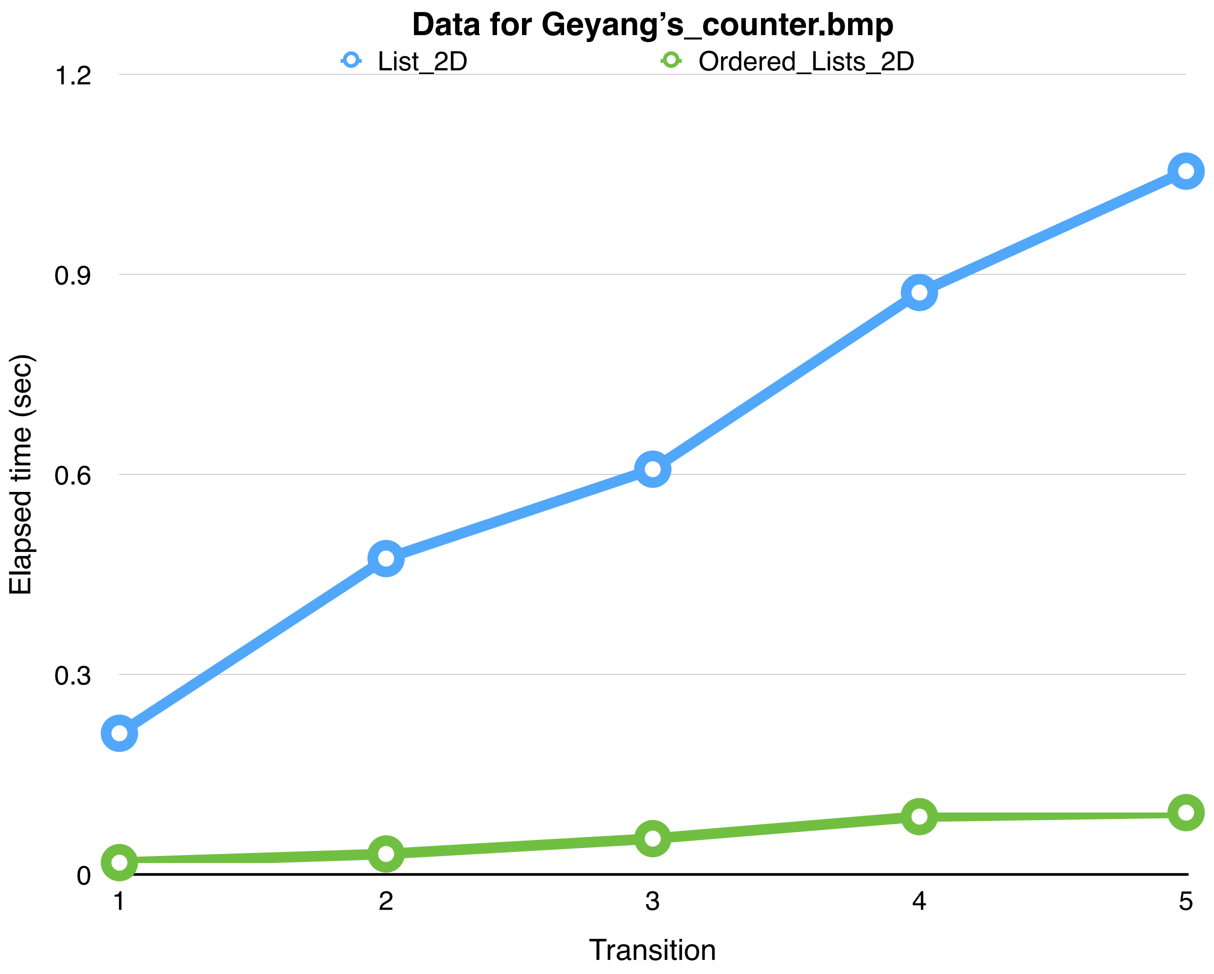
From the above diagram, we can see that on an average input, Ordered_Lists_2D model is about **3 times faster** than List_2D model.

The second one (12 cells) is a simple clock Wireworld (Cycle.bmp), which is a simpler case (very close to the best case) for these two programs in my opinion.



From the above diagram, we can see that at the very simple cases, Ordered_Lists_2D is not so smart as List_2D, this is because Ordered_Lists_2D may over-complicate some simple problems.

The third one (3515 cells) is my own Wireworld (Geyang's_counter.bmp), which is a more complicated case for these two programs in my opinion. (Originally I want to use Langton's_Ant_3x3.bmp however the List_2D just cannot run it)

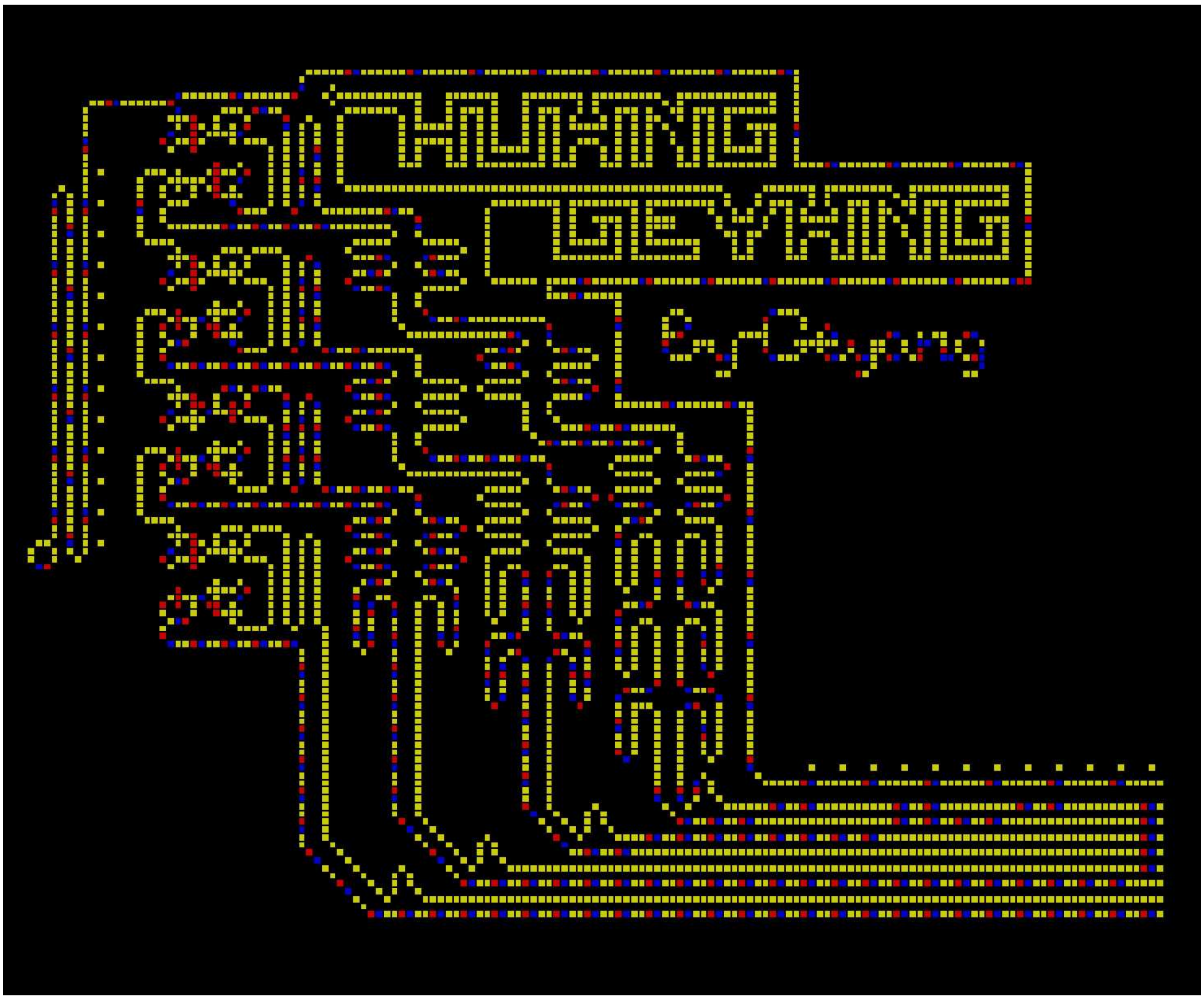


From the above diagram, we can see that Ordered_Lists_2D works very well in the extreme complicated case, the speed of Ordered_Lists_2D is about 10 times faster than that of List_2D. This is because for the large bmp the size of List_2D's searching space is about n (the length of the side of bmp) times of that of Ordered_Lists_2D. This results in the computational complexity of List_2D growing much faster than that of Ordered_Lists_2D.

Possible Improvement

Since the only thing we concern in the bmp is the conductor and the electron on it, I am thinking that maybe we can construct a new data structures to get a more efficient algorithm. If we create a linked table of the **Conductor**s (so every **Conductor** can be representing by a hash value based on its coordinates), and a list of current electrons on the graph, then we can simulate the Wireworld in a new way.

Extension



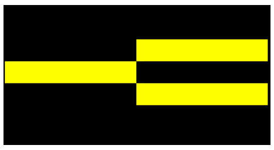
I would proudly present my own Wireworld to you, Geyang's counter, which is a 4-cycle binary counter in Wireworld.

This time, my task is programming in Wireworld, which is very interesting. I still design based on the two principle, accuracy and efficiency. The accuracy here is to provide the correct binary representation for the count of the 4-cycle signals, and it is the basic requirement for the program. The efficiency here is not like the normal programs. Since the Wireworld is simulated by my previous program, I knew that the efficiency of the Wireworld is inversely proportional to the size of Wireworld. Why? Larger bmp size means more cells in the Wireworld, and hence more time taken for each transition, therefore the time between the input and output in the Wireworld is longer, which means lower efficiency. As a result, I need to minimise my Wireworld and use as few cells as possible. On top of these, I try to make my Wireworld to be easy to understand. This might be a little bit hard, but I think users will understand this with my little help in explanation.

Developing Geyang's_counter.bmp

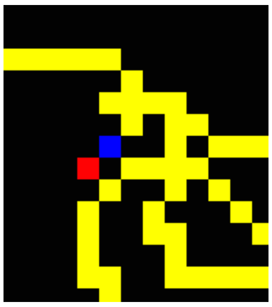
Step 1 : Copy the signal from input

We need to keep the original signal as it will be shown in the result. Therefore we copy another signal for processing.



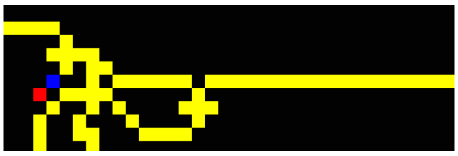
Step 2 : Construct a flip-flop circuit for 4 cycle

Flip-flop circuit is the key component of the binary counter unit.

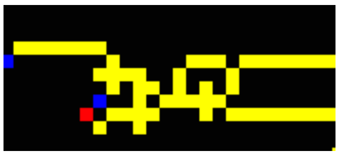


Step 3 : Construct a detourer for the flip-flop

I name it detoured is because it detours a copy of a beam electron by one cycle so it can meet the original beam by delaying one cycle, and magic happens!



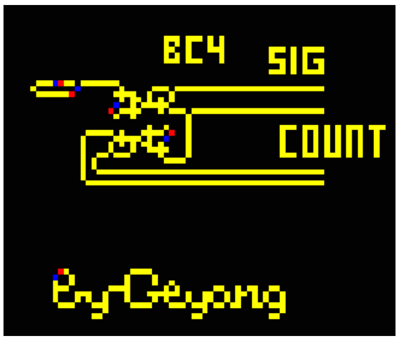
Original design of FF4 with detourer



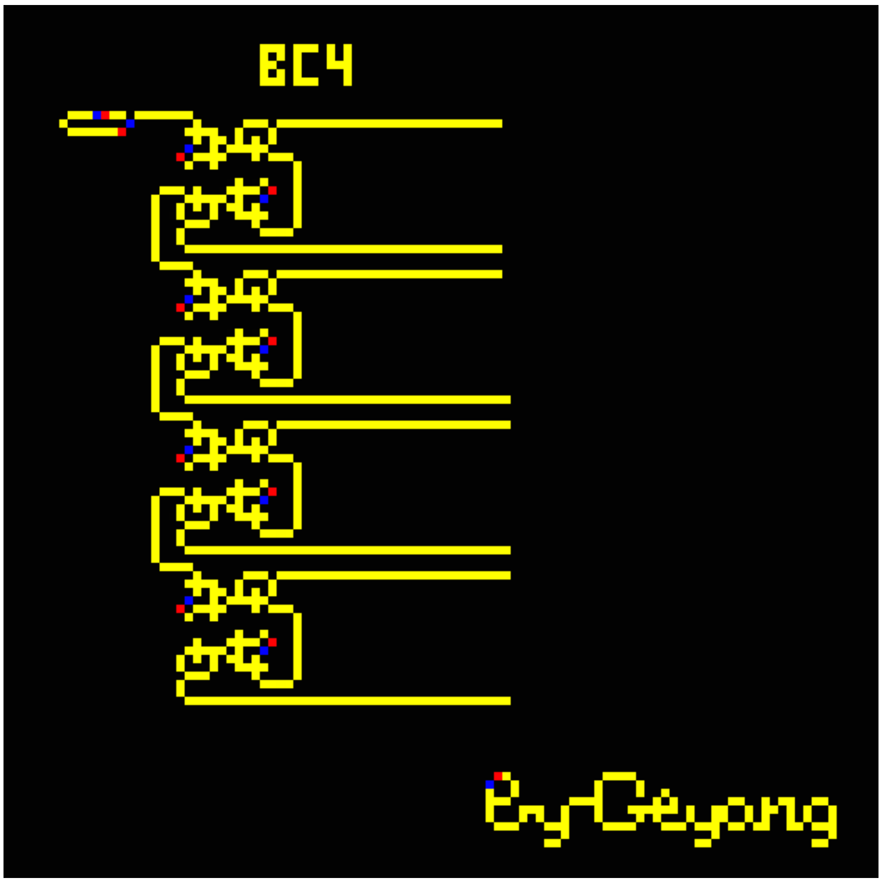
Modified design

Step 4 : Compress two single unit together to minimise the cell used

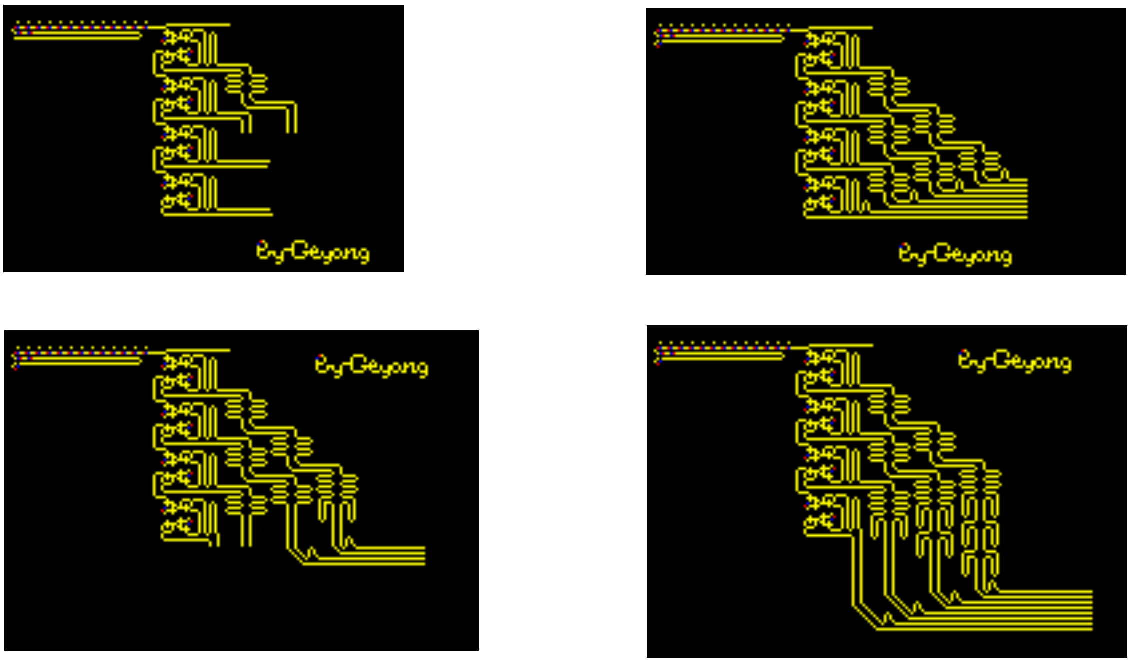
One single unit = one flip-flop + one detourer



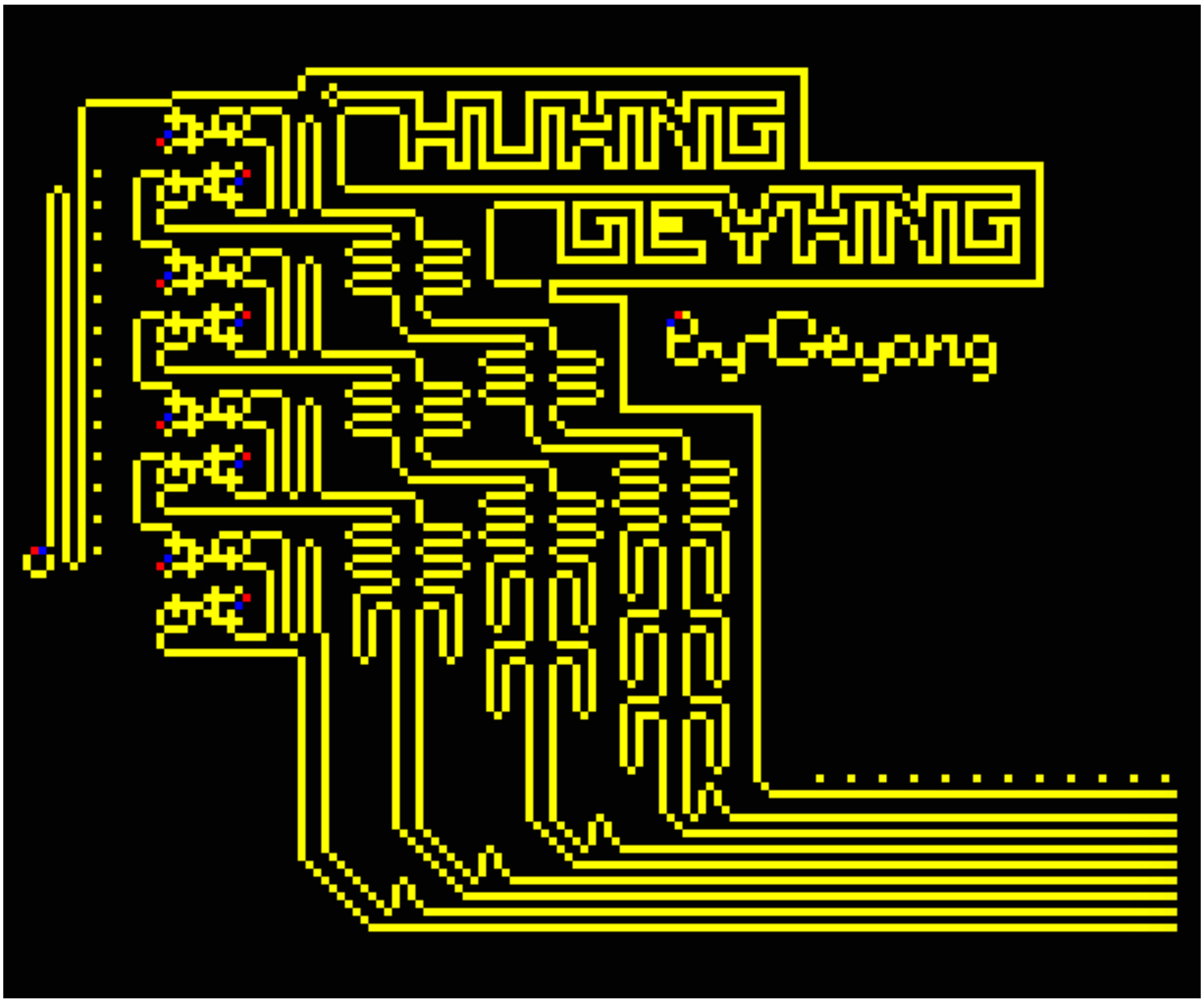
Step 5 : Repeating 8 units, to form a 8 bit binary counter



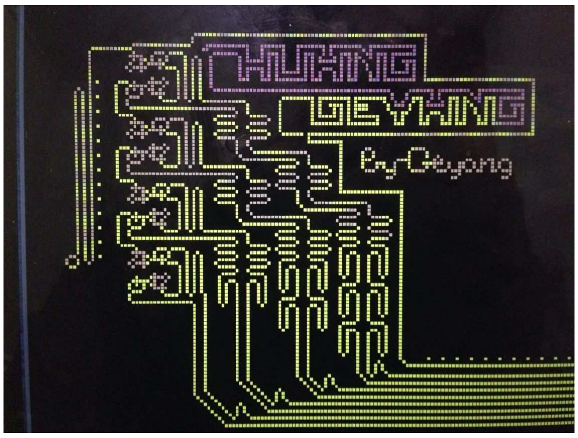
Step 6 : Reclocking the output from each unit
(The wiring patterns also have meanings in them)



Finally, we finish!



I actually put a small design on my name, as you can see the left top of my name there is a clock, at image above it is being turned off. If we turn on the clock in the initial state, my name will actually become a “progress bar”! So once the electron reaches the end of my name, the last electron from the input bar (or the first electron from the 8-cycle clock) at the left reaches the end of the output bar. This basically means the time taken for my Wireworld to complete the counting task is roughly equal to the time taken for a electron to complete a travel of my name.



Interesting thing happens at —fps=60, my name is filled up with purple.
(Use 3-cycle clock)

Little Summary

This assignment is so interesting. For me, it is far more than just an assignment. I learn the term “cellular automata” during this assignment, and it introduces a new field to me. In cellular automata, the program is probably possible to produce new more complicated program in my opinion. I think I may modified the whole Wireworld program to create some new cellular automata with my own rules. The most interesting parts are optimising algorithms for Ordered_Lists_2D and designing my own Wireworld, and I spend the most time within these two areas. After all, I hope my programs will bring you aesthetic enjoyment.