AC21009 - Computer Systems 2A

Coursework 2 - Cellular Automaton

This development project is designed to give you a simple and straightforward, but fun, initial project to develop in C/C++ under Linux. Several of the concepts you will use in this coursework will be used in your next (larger) coursework, so this will give you a chance to develop and implement them in advance of that coursework.

Cellular Automata

Generally a cellular automaton (CA) is a regular grid of cells. Each cell can exist in one of a finite number of states. The grid can be in any finite number of dimensions but we are only really concerned with 1- (and possibly 2-) dimensional cellular automata. Time is considered to be **discrete** rather than continuous. This means that the whole automaton exists in a single state at a given time, e.g. at time t=1, and then according to the rules, the automaton changes to another state at time t=2, and so on. The state of any given cell at time t is a function of the states of a finite number of cells (called its *neighbourhood*) at time t-1. This means that the next state of a given cell is determined by the states of its neighbours in the immediately previous time step. These neighbours are a selection of cells relative to the specified cell, and do not change (though the cell itself may be in its neighbourhood, it is not usually considered a neighbour). Every cell has the *same rule* for updating, based on the values in its own neighbourhood. Each time the rules have been applied to the whole grid, a **new generation** is created.

Background Reading

Wikipedia has a good introduction to CAs at:
`http://en.wikipedia.org/wiki/Cellular_automata`

Most popular science books on chaos theory will include a chapter or two on CAs. However a good book is ”A New Kind of Science” by Stephen Wolfram (2000) which may be worth dipping into for some added background - it is available online at:
`http://www.wolframscience.com/`

You can find out more about CAs at:
`http://mathworld.wolfram.com/CellularAutomaton.html`

Searching YouTube for cellular automata yields lots of movies and animations of cellular automata in action, as well as some documentaries of scientists talking about their implications.

Finally, you should also consider getting a book on C programming from the University Library. There are many to choose from and most introductory texts will be sufficient at this stage to provide you with extra support whilst doing your development projects.

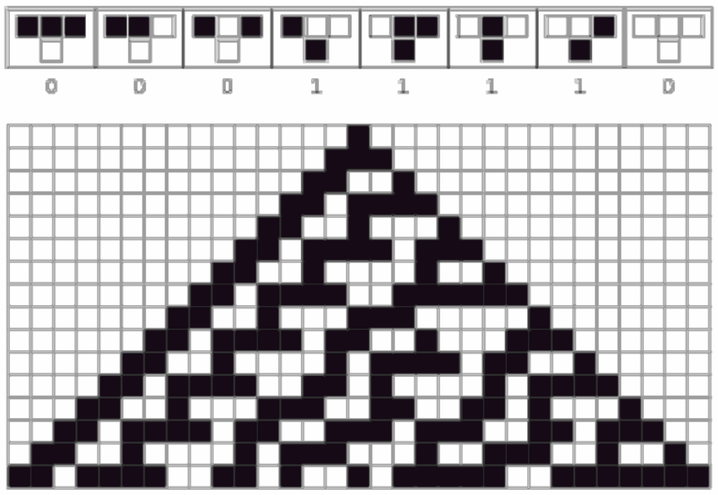
Requirements

Your task is to create a simple 1-dimensional (1-D) cellular automaton (CA) which will display each generation of your automaton in the shell (**as this exercise is to give you practice C programming in a Unix-like environment, it should be performed using these tools in the lab, or equivalent**). Our CA will be binary, so each cell will display *either 1 or 0*.

In a 1-D CA, each generation is just a *single line*. The values of the cells in each generation are determined by applying the rules to the values of the cells in the *previous* generation. For example, if we use rule set 30 (remembering that 30 = 00011110 in binary), gives:

Current state of 3 cells above	`111`	`110`	`101`	`100`	`011`	`010`	`001`	`000`	
New state for centre cell	**`0`**	**`0`**	**`0`**	**`1`**	**`1`**	**`1`**	**`1`**	**`0`**	($=30_{10}$)

then we should get output similar to the following (although yours does not actually have to look exactly like this):



Getting Started

You should probably start with storage for **two generations** of cells: a parent generation and a child (or current) generation. One way of implementing this storage is using **arrays**. Initially you will seed the parent generation at time t=0, i.e. put the value 1 or 0 into each cell of the parent array. You now need to work out what the child generation should contain given the contents of the parent. You do this by applying the rules as discussed above, for example, if the parent cell and its neighbours to either side contain the values 0, 1, 1, then the child cell should contain the value 1. Once you have completed the child generation you can discard the parent, although you might want to consider writing it to a file so that you can compare the output between different runs. At this point the **child generation becomes the new parent generation**, and you should work out what the *next* child generation should be. You might want to start with rule set 30, as shown above, for specifying how the child generation is derived from the parent. Alternatively, you might want to put together *your own rule set* (there are 256 possible different rule sets that you can generate from various combinations of the 3 ancestor cells, and this is only for the most simple type of automata).

Before you do this you should decide how big you want the storage for each of your generations to be, e.g. how many cells make up a generation? 10? 50? 100? Too few and you might have insufficient space for interesting patterns to emerge, too large and you will not be able to fit much of any given generation into your terminal window.

Rather than spending your time learning to do graphics in C/C++ (although this might be a good extension once you have the basics working, because graphics give you more resolution for the output) you should consider how you could indicate the different cells in the terminal, for example, using a 1 to indicate black and a 0 (or even possibly a space) to indicate white. One other thing that you will have to consider is how to store your automata data whilst you are working on it (e.g. the previous generation and current generation). You might want to use an array of INTs to begin with, so that each integer in the array is a cell in the automata, but there are much more efficient ways to store the cells if you consider that each cell has only two states and is therefore very similar to a bit (binary integer). You could store your data very efficiently as bit arrays but that would limit you to only two states for your cells and thus your code would be limited to the very simplest automata. Perhaps a happy medium might be to store each cell as a CHAR because a CHAR generally uses about half as much memory as an INT? Perhaps however there is more data that you would like to store with each cell and therefore some sort of simple STRUCT would be better?

To get the very highest marks possible for this project, you should consider implementing some of the extensions to the basic requirements. There are many extensions that you could implement if you have sufficient time. For example, you might consider the following:

Simple Extensions - RECOMMENDED

To perform well in this assignment, your solution ***might*** include the following features:

- conversion of binary strings e.g. “101001010101” into an array of variables (binary or integers) and back again
- conversion of binary numbers into their decimal values and back again
- saving the output of your program to a text file on disk (and loading saved files)

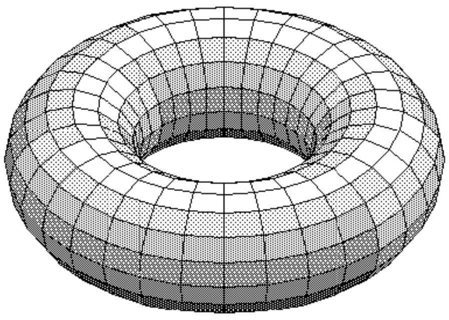
Note that future assignments ***will*** require you to include the features above, so gaining these skills ***now*** will be useful when working on these later assignments.

Other Simple Extensions

- Investigate rules other than rule 30 given above and compare the output that you get - maybe you could get your program to *randomly* generate the rule set at the start.
- Allow the user to select different lengths of the first generation, and to select different numbers of generations to produce.
- Allow the user to select, or create, the first generation
- Wrap the ends of your automata array around so that the right-hand end continues onto the left-hand end and vice-versa.
- Write your automata out to a text file so that you can look at the results once the program has completed.

Further extensions

- Implement a version of Conway's Game of Life (which is basically a 2-dimensional cellular automaton with a particular set of rules).
- Develop a different 2-dimensional automaton.
- Try wrapping around *all of the sides* so that it forms a 3-dimensional torus as shown below:



Submission & Assessment
This task must be carried out *in pairs*, and you *must* submit a peer assessment form giving the contribution of both team members.

This coursework is due for submission via MyDundee (under AC21009 Assignments) at **12 noon on Friday 2nd November 2017** (Friday of Week 8) and is worth 10% of your total grade for this module. Keep a copy of what you submit in case there are problems with your submission. As well as the source code for your management system, you should include a report of 550-600 words which explains how you approached the assignment, the problems that you faced, and the solutions that you have created - see the **report writing guidelines** on MyDundee for more details on what is expected in the report.

Your submission should be a single ZIP (*not RAR*) file containing:

- the **source code** for your Cellular Automaton, ready to be compiled under Linux
- your **550-600 word report** (as a Word or LibreOffice document, with word count given) **Your report should also clearly state which Linux C/C++ compiler you used for your project.**
- your **peer assessment** form (for yourself, or for both members of your team)

All elements should include your name and matriculation number**.**

Your software should run on the QM Lab PCs, as these will be used when marking, and should **not** require the use of any special code libraries which are not normally available on these machines.

Marking Guidance

A **minimal solution** which outputs a clear text-based representation of *one rule only*, with *no user options*, will achieve **30 marks** (out of 85).
An **extended solution** which includes **all** of the simple extensions will achieve *up to* **73 marks** (out of 85).
Implementing the **further extensions** will additionally achieve *up to* **12 marks**.

Marks will be lost for a poor or incomplete display, incomplete functionality and poor or incomplete coding. Extra marks will be given for outstanding implementation and for including any of the extensions suggested, or other relevant features.

A well-written **report** which meets all the given guidelines will achieve up to **15 marks**.

A **minimal solution with a poor report** will thus achieve around 40% (D3), a full **extended solution with an adequate report** will thus achieve up to 85% (A3); adding the **further extensions** could take this to an A1.