CSE 232 Spring 2017

Programming Project #6

Assignment Overview

This project focuses vectors and their use. It is worth 50 points (5% of your overall grade). It is due Monday 10/23 before midnight

The Problem

A *cellular automaton* (see https://en.wikipedia.org/wiki/Cellular_automaton) is specialized approach to computing. It consists of a set of *cells*, each of which is in either an on or off state, symbolized by the values 1 or 0 respectively. These cells can change their state over time based on a set of update *rules*. The change of states of the cells can be used to do a kind of computing. Each cell is updated during each iteration of the algorithm, potentially changing the state of all the cells in the automaton.

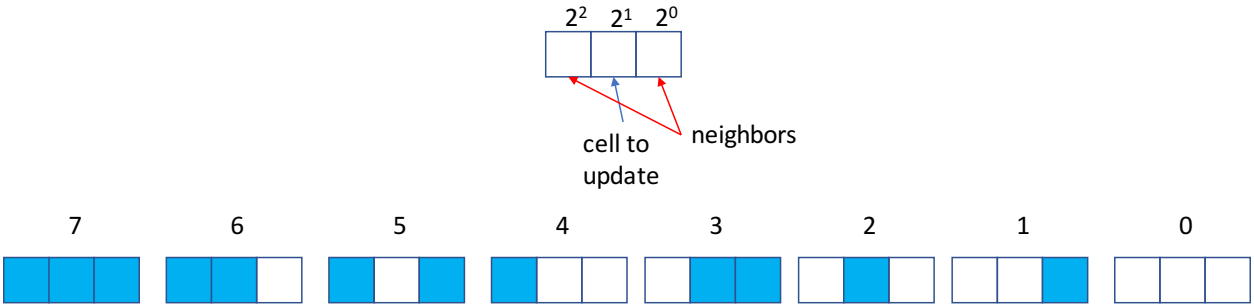
The Game of Life

The most popular cellular automaton you might have heard of is called "The Game of Life" (https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life) invented by the English mathematician John Conway in 1970. It is played on a two-dimensional grid cells and the game-of-life update rule provides for a very complex, and entertaining, development of cell states over time.

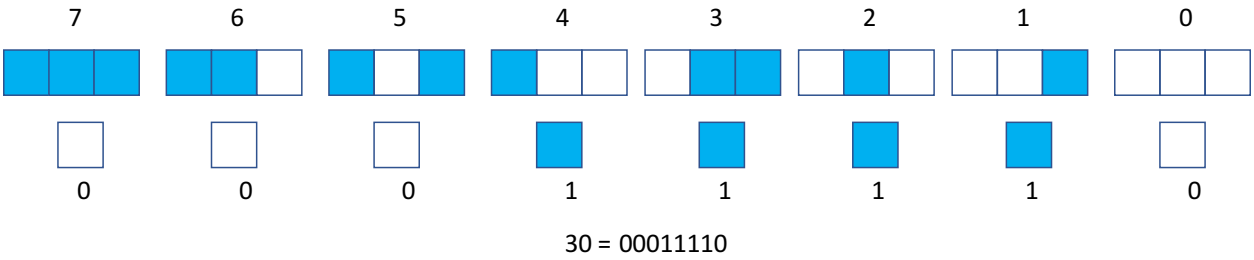
Wolfram 1D cellular automata

Stephen Wolfram is an English mathematician/scientist who founded Wolfram, maker of Mathematica and Wolfram Alpha. He quantized the concept of a 1D cellular automaton as discussed here (http://mathworld.wolfram.com/CellularAutomaton.html). The basic idea is this.

The simplest version of a 1D cellular automaton consists of the central cell, whose state might change during an iteration, and its two neighbors. We can construct 8 possible rules for update of a cell based on the value of the central cell and its two neighbors. We label the rule based on a binary enumeration of the values of the 3 cells

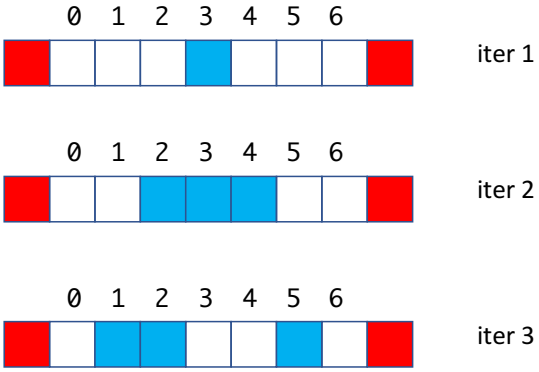


These are the possible patterns, but we now need to provide how the central cell changes for each of the 8 rules. We can do that by specifying *rule number*, an 8-bit number, range 0-255, that labels how the central cell should change for each rule based on a binary representation, in 8 bits, of that rule number. Consider the rule number 30 (used in the wolfram example). Its binary representation is 00011110. We mark below each of the 8 the rules the binary value of the rule number. This binary value indicates, if that rule applies, the value (0 or 1) the central cell takes on in the next iteration.



Iteration

For rule number 30 and a vector of length 7 starting with all the elements set to 0 except the element at index 3, let's follow the iteration of rule application. ***Important note***: Because we need 3 cells for our rules to apply, we artificially provide two end elements of ***value 0*** for every vector. This means that the first and last indices can be calculated according to the listed rules. We indicate this in the vector below as the red elements, end caps that are fixed at 0. Note they **don't have to exist**, they can be assumed to be there programmatically.



Let's look at the first iteration. We apply the rules to each index of the `iter 1` line generating the results to the `iter 2` line.

- index 0 → rule 0 applies, index 0 is white (0) next iteration
- index 1 → rule 0 applies, index 1 is white (0) next iteration
- index 2 → rule 1 applies, index 2 is blue (1) next iteration
- index 3 → rule 2 applies, index 3 is blue (1) next iteration
- index 4 → rule 4 applies, index 4 is blue (1) next iteration
- index 5 → rule 0 applies, index 5 is white (0) next iteration
- index 6 → rule 0 appliex, index 6 is white (0) next iteration

We then reapply the rules to the `iter 2` line generating `iter 3`, then reapply the rules to the `iter 3` line generating `iter 4`, etc. until the iterations end.

Your Tasks

Complete the Project 6 by writing code for the following functions. Details of type for the functions can be found in `proj06_functions.h` (provided for you, see details below):

Functions

function `to_binary`
- single `int` argument
- returns a `string`
- converts the argument to a binary string that is 8 chars long consisting of either `'0'` or `'1'`
- **Error Check**: if the argument is not in the range 0 – 255, returns a string of 8 `'0'` (binary 0)

function `next_val`
- two arguments
  - `const vector<int> &v`
  - `const string &rule_binary_string`
- returns a `int`
- v is assumed to be of size 3 (no error checking). It applies the rule number, represented by the string `rule_binary_string`, and returns the new central value
- No error checking

function `one_iteration`
- two arguments
  - `const vector<int> &v`
  - `const string &rule_binary_string`
- returns a `vector<int>`
- creates a new `vector<int>` (the return value) by applying `next_val` to each index of the argument vector `v` using the rule number indicated by `rule_binary_string.`
- No error checking.

function `print_vector`
- two arguments
  - `const vector<int> &v`
  - `ostream &out`
- prints the contents of `v` to `out`:
  - format is number followed by a comma for each element
  - no trailing comma at the end (last element has no following comma).
  - look at the Mimir test case

function `read_vector`
- two arguments, `vector<int> &v` which is ***empty*** and `string fstring`, a file name to open
- no return
- an input file is used that consists of a set of binary values, 0 or 1, on the same line, each separated by a space: 0 0 0 1 0 0 0 for example
- fills the argument `v` with the provided values. Size of `v` will be the number of elements read in.
- **Error checks**:
  - if v is not empty, no values are read and v remains unchanged
  - if fstring cannot be opened, no values are read and v remains unchanged

Deliverables

`proj06/proj06_functions.cpp` -- your completion of the functions described above. Only `proj06/proj06_functions.cpp` is turned in to Mimir.

1. Remember to include your section, the date, project number and comments.
2. Please be sure to use the specified directory and file name.

Test Cases

We are changing the test cases up a bit. There are 5 pairs of test cases, 1 for each function. The first of each pair is visible as always. That is, you can see the results of testing. The second, however, is not fully visible. You can see if you pass the test but you cannot see the input/output values of the test. This is for two reasons:

1. You need to do testing on your own to see how your code runs. Blindly submitting to Mimir is not a viable strategy going forward.
2. Too many people are simply writing code that passes the test but doesn't address the problem. This will help with that.

Manual Grading

In lab05 you worked with Code Style. The TAs will apply the things that you learned in your Code Review lesson on the code you submit. 4 points are reserved (out of 50) for this.

Assignment Notes

1. Mimir allows us to test the functions in `proj06_functions.cpp` individually and we will do that.
2. You can write your own `pro06_main.cpp` to test your code, but only `proj06_functions.cpp` is turned into Mimir
3. `proj06_functions.h` is provided in the project06 directory. It will be used in testing. When we do testing, we will use this file. If you change and submit your own version, Mimir will ignore it and use ours.