School of Computing and Information Systems

comp10002 Foundations of Algorithms
Semester 2, 2024
Assignment 2

Learning Outcomes

In this project, you will demonstrate your understanding of dynamic memory and linked data structures (Chapter 10) and extend your program design, testing, and debugging skills. You will learn about cellular automata, implement a tool for performing cellular automata computations, and use this tool to solve a practical problem.

Background

A *cellular automaton* is a model of computation composed of *cells*, each in one of a finite number of *states*. For each cell, a set of cells called its *neighborhood* is defined. At the beginning of computation (time `t=0`), each cell is assigned a state. The states of all its cells determine the state of the automaton. The automaton progresses from its current state at time `t` to the next state at time `t+1` by updating the states of its cells using an update *rule*. This rule defines a relation between the neighborhood of a cell at time `t` and the state of the cell at time `t+1`. The rule is typically deterministic and is applied for each cell in the current state simultaneously.

In an *elementary cellular automaton*, cells are arranged in a one-dimensional array and each cell is in `0` or `1` state denoting *on* and *off* cells, respectively. An automaton of 31 cells, with cells at positions 0, 1, 12, 15, 18, and 29 on (counting from left to right starting from zero) and the other cells off, is shown below.

```
11000000000010010010000000000010
```

The neighborhood of a cell in an elementary cellular automaton comprises the cell itself and its two immediate neighbor cells. The update rule of an elementary automaton maps such possible neighborhoods of a cell to its states in the next time step. One such update rule is depicted below.

```
000 001 010 011 100 101 110 111
 0   1   1   1   1   0   0   0
```

The top row lists all eight possible neighborhoods of a cell, with the cell in the middle being the cell of interest and the other two cells being its neighbors. For example, the `010` neighborhood denotes that the cell is on and both its immediate neighbors are off, whereas `110` is the neighborhood of an on cell with its left neighbor on and right neighbor off. The bottom row shows the state of the cell in the next time step for the corresponding neighborhoods. For instance, a cell with neighborhood `010` stays on (`1`) in the next step, while a cell with neighborhood `110` switches off (`0`) in the next step. A *run* of the first three computation steps of the example automaton using the example update rule is listed below; numbers at the start of each line denote time steps.

```
0:  11000000000010010010000000000010
1:  10100000000111111111000000000110
2:  10110000001100000000100000011100
3:  10101000011010000001110000011011
```

The gray background highlights eight applications of the update rule on cells at different positions at different time steps. For example, the neighborhood of the cell at position 12 at `t=0` (`010`) determines the state of this cell at `t=1` (`1`). As another example, the neighborhood of the cell at position 24 at `t=0` (`000`) determines that this cell stays off at `t=1` (`0`). When determining the neighborhoods of the edge cells (first and last cells in the array), the array is "wrapped" around. Specifically, the cell after the last one is the first cell, and the cell before the first one is the last cell. Hence, the neighborhood of the last cell at `t=0` (position 30) is `101` determining that the the cell at position 30 at `t=1` is off. Once cell states at time `t` are determined, one can compute cell states at time `t+1`.

There are $8 = 2^3$ possible neighborhoods and $256 = 2^{2^3}$ possible update rules defined for elementary cellular automata. It is convenient to refer to each rule by its *code*, given as a number from 0 to 255. If one traverses possible neighborhoods in order `111`, `110`, `101`, `100`, `011`, `010`, `001`, `000` (in the descending order of the encoded binary numbers), writes out the corresponding next states in the same order, and interprets the obtained sequence of symbols as the binary representation of an integer, the resulting number is the code of the rule. For instance, the example update rule presented above results in the binary representation `00011110` of number 30. Hence,

this update rule is rule 30. Many of the 256 rules are trivially equivalent, for example, by swapping roles of `0` and `1` or by reflection via the vertical axis. There are only 88 fundamentally different update rules.

Cellular automata have many applications, including studies of complex behaviors in nature (rule 30), prime numbers (rule 90), traffic flows (rule 184), and computability (rule 110). Artem and Alistair need a tool to study new ways of solving computational problems using cellular automata. Your task is to implement this tool.

Input Data

Your program should read input from `stdin` and write output to `stdout`. The input will list instructions, one per input line, each terminating with "\n", that is, one newline character. The input specifies how to configure and execute an elementary cellular automaton and to collect basic statistics about the evolution of its states. The following file `test0.txt` uses seven lines to specify an example input to your program.

```
1  31
2  30
3  **.........*..*..*.........*.
4  10
5  0,5
6  30,10
7
```

Lines 1–3 in the `test0.txt` file specify a configuration of the automaton. Line 1 defines the size of the automaton, that is, the number of cells in the array. Line 2 specifies the code of the update rule, whereas line 3 encodes the cell states, with "*" encoding on states and "." encoding off states. Hence, line 3 specifies the example automaton discussed in the Background section. Line 4 specifies the number of time steps to evolve the automaton in Stage 1 of the program. Finally, lines 5 and 6 are instructions to analyze the evolution of states of cells at specific positions in the automaton in Stages 1 and 2 of the program, respectively.

The input will always follow the proposed format. You can make your program robust by handling inputs that deviate from this format. However, such extensions to the program are not expected and will not be tested.

Output

The output your program should generate for the `test0.txt` input file is in the `test0-out.txt` file and below.

```
1   ==STAGE 0============================
2   SIZE: 31
3   RULE: 30
4   ------------------------------------
5    000 001 010 011 100 101 110 111
6     0   1   1   1   1   0   0   0
7   ------------------------------------
8      0: **.........*..*..*.........*.
9   ==STAGE 1============================
10     0: **.........*..*..*.........*.
11     1: *.*........*********........**.
12     2: *.**......**........*......**..
13     3: *.*.*....**.*......***....**.**
14     4: ..*.**..**..**....**..*..**..*.
15     5: .**.*.***.***.*..**.******.****
16     6: .*..*.*...*...****..*......*...
17     7: *****.**.***.**...****....***..
18     8: *.....*..*...*.*.**...*..**..**
19     9: .*...******.**.*.*.*.*****.***.
20    10: ***.**......*..*.*.*.*.....*..*
21  ------------------------------------
22  #ON=3 #OFF=3 CELL#0 START@5
23  ==STAGE 2============================
24  RULE: 184; STEPS: 14.
25  ------------------------------------
26    10: ***.**......*..*.*.*.*.....*..*
27    11: **.**.*......*..*.*.*.*.....*.*
28    12: *.**.*.*......*..*.*.*.*.....**
29    13: .**.*.*.*......*..*.*.*.*....**
30    14: **.*.*.*.*......*..*.*.*.*...*.
31    15: *.*.*.*.*.*......*..*.*.*.*...*
32    16: .*.*.*.*.*.*......*..*.*.*.*..*
33    17: *.*.*.*.*.*.*......*..*.*.*.*..
34    18: .*.*.*.*.*.*.*......*..*.*.*.*.
35    19: ..*.*.*.*.*.*.*......*..*.*.*.*
36    20: *..*.*.*.*.*.*.*......*..*.*.*.
37    21: .*..*.*.*.*.*.*.*......*..*.*.*
38    22: *.*..*.*.*.*.*.*.*......*..*.*.
39    23: .*.*..*.*.*.*.*.*.*......*..*.*
40    24: *.*.*..*.*.*.*.*.*.*......*..*.
41  ------------------------------------
42  RULE: 232; STEPS: 15.
43  ------------------------------------
44    24: *.*.*..*.*.*.*.*.*.*......*..*.
45    25: .*.*....*.*.*.*.*.*..........*
46    26: *.*......*.*.*.*.*............
47    27: .*.......*.*.*.*.............
48    28: ...........*.*.*.............
49    29: ............*.*..............
50    30: .............*...............
51    31: ............................
52    32: ............................
53    33: ............................
54    34: ............................
55    35: ............................
56    36: ............................
57    37: ............................
58    38: ............................
59    39: ............................
60  ------------------------------------
61  #ON=10 #OFF=20 CELL#30 START@10
62  ------------------------------------
63    10: ***.**......*..*.*.*.*.....*..*
64  AT T=10: #ON/#CELLS < 1/2
65  ==THE END============================
66
```

Stage 0 – Reading, Analyzing, and Printing Input Data (10/20 marks)

The first version of your program should read the automaton configuration from input, construct a representation of the initial state of the automaton at `t=0`, and print basic information about the automaton to the output. The first eight lines in the `test0-out.txt` file correspond to the output your program should generate in Stage 0 based on the first three input lines in the `test0.txt` file. Line 1 of the output prints the Stage 0 header. Lines 2 and 3 print the size of the automaton and the code of the update rule, respectively. Lines 5 and 6 visualize the update rule using the principles explained in the Background section. Line 8 prints the automaton state at `t=0`. Use `"%4d:"` format specifier to `printf` the time step, and use "`*`" and "`.`" characters to `printf` on and off cell states, respectively. Finally, lines 4 and 7 print the delimiter lines of 37 "`-`" characters.

You should not make assumptions about the maximum number of cells requested from the input for the automaton your program should construct. Use dynamic memory and data structures of your choice, for example, arrays or linked lists, to store the history of states of the automaton.

Stage 1 – Execute Automaton (16/20 marks)

In Stage 1, your program should execute the automaton configured in Stage 0 for the requested number of time steps and print the number of on and off states a particular cell has been at in the requested period of execution. Lines 9 to 22 in the `test0-out.txt` file correspond to the output your program should generate in this stage based on the instructions at lines 4 and 5 in the `test0.txt` input file. Line 4 in the `test0.txt` file specifies the number of time steps the automaton should execute starting from the initial state (`t=0`), while line 5, using format `x,y`, requests to count and print the number of on and off states of the cell at position `x` in the automaton starting from time step `t=y` and to the last so far executed time step. Hence, the instructions in the `test0.txt` file request to evolve the initial state of the automaton for ten time steps and count the number of observed on and off states of the cell at position zero starting from time step `t=5` up to and including time step `t=10`.

The output of Stage 1 should start with the corresponding header; see line 9 in the `test0-out.txt` file. The subsequent lines, lines 10 to 20 in the `test0-out.txt` file, print the automaton states in chronological order they were observed, starting with the initial state (`t=0`) to the last computed state (`t=10`). Again, use "`*`" and "`.`" characters to denote on and off cell states and the `"%4d:"` format specifier to format the output of time steps.

The output line that follows all the printed states should be the delimiter line of 37 "`-`" characters; see line 21 in the `test0-out.txt` output file. Next, your program should print the counts of on and off states of the requested cell using the `"#ON=%d #OFF=%d CELL#%d START@%d\n"` format specifier; see line 22 in the output.

You should not make assumptions neither about the maximum number of time steps that can be requested to evolve the automaton nor about the requested cell position or time step to start counting the cell states.

Stage 2 – Solve Practical Problem (20/20 marks)

Given an array of zeros and ones (bits), the *density classification* problem consists in deciding if the array contains more zeros or more ones. Fukś [1] showed that elementary cellular automata based on rules 184 and 232 could solve this problem perfectly. Specifically, given an automaton of k cells, one should first execute the automaton for $n = \lfloor (k-2)/2 \rfloor$ time steps using update rule 184 and then evolve the resulting state for $m = \lfloor (k-1)/2 \rfloor$ steps using update rule 232; $\lfloor x \rfloor$ denotes the largest integer less than or equal to x. If all the cells of the resulting automaton are on, then the state of the original automaton has more on states. In contrast, if all the cells of the resulting automaton are off, then the state of the original automaton has more off states. Finally, if cells in the resulting automaton exhibit an alternating sequence of on and off states (regardless of the state of the cell at position zero in the array), then the state of the original automaton has the same number of on and off cells.

In Stage 2, your program should apply the procedure of Fukś described above to solve the density classification problem for the state of the automaton computed for the last time step in Stage 1 (`t=10` for the `test0.txt` input file). The output of Stage 2 should start with the corresponding header; see line 23 in the `test0-out.txt` file. In the subsequent lines, your program should print the evolution of the automaton using rules 184 and 232, with rules applied in the correct order for the correct number of time steps. Lines 24 to 60 in the `test0-out.txt` file print this output for the automaton resulting from Stage 1. Lines 24 and 42 inform about the changes in the update rule and the number of time steps the new rules are applied, while lines 25, 41, 43, and 60 are the delimiter lines of 37 "`-`" characters.

The subsequent line (line 61 in `test0-out.txt`) in the output should print the number of on and off states for the cell at position starting from the time step requested on line 6 in the `test0.txt` input file. Finally,

after the delimiter line (line 62 in `test0-out.txt`), your program should print the automaton for which the density classification problem was solved (line 63) and the answer to the problem (line 64). As all the cells in the automaton at `t=39` are off, there are more off cells than on cells in the automaton at `t=10`. Replace the "`<`" symbol in the result message (line 64) with "`=`" or "`>`", as required by other inputs to your tool.

Stage 3 – Have Fun (no marks for fun as fun is your reward)

Search for other solutions to interesting practical problems using elementary cellular automata, implement them, and share them with the class via the discussion board.

References

[1] Henryk Fukś. Solution of the density classification problem with two cellular automata rules. *Physical Review E*, 55(3), 1997.

Pay Attention!

Two further test inputs and outputs are provided. The outputs generated by your program should be *exactly the same* as the sample outputs for the corresponding inputs. Use `malloc` and dynamic data structures of your choice to store automata, rules, etc. Before your program terminates, all the `malloc`'ed memory must be `free`'d.

Important...

This project is worth 20% of your final mark, and is due at **6:00pm on Friday 11 October**.

Submissions made after the deadline will incur penalty marks at the rate of two marks per (part or all) working day. Students seeking extensions for medical or other "outside my control" reasons must lodge a request following the FEIT process linked from the LMS page once those circumstances arise. If you attend a GP or other health care service as a result of illness, be sure to obtain a letter from them that describes your illness and their recommendation for treatment. Suitable documentation should be attached to **all** extension requests.

You need to submit your program for assessment via **Gradescope (Assignment 2)**. Submission is **not possible through Grok**. Multiple submissions may be made; only the last submission that you make before the deadline will be marked. If you make any late submission at all, your on-time submissions will be ignored, and if you have not been granted an extension, the late penalty will be applied.

As you can submit your program multiple times, test your program in the test environment early. The compilation in the Gradescope environment may differ from Grok and your laptop environment. Note that Gradescope may overload and even fail on the assignment due date when many students submit their programs, and if that does happen, it will not be a basis for extension requests. *Plan to start early and to finish early!!*

A rubric explaining the marking expectations is linked from the LMS. Marks will be available on the LMS approximately two weeks after submissions close, and feedback will be made available via Gradescope.

Academic Honesty: You may discuss your work during your workshop, and with others in the class, but what gets typed into your program must be individual work, not copied from anyone else. So, do **not** give hard copy or soft copy of your work to anyone else; do **not** have any "accidents" that allow others to access your work; and do **not** ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest". The best way to help your friends in this regard is to say a very firm "**no**" if they ask to see your program, pointing out that your "**no**", and their acceptance of that decision, are the only way to preserve your friendship. See `https://academicintegrity.unimelb.edu.au` for more information. Note also that solicitation of solutions via posts to online forums, whether or not there is payment involved, is also Academic Misconduct. In the past students have had their enrolment terminated for such behavior.

The assignment page contains a link to a program skeleton that includes an Authorship Declaration that you must "sign" and include at the top of your submitted program. Marks will be deducted (see the rubric) if you do not include the declaration, or do not sign it, or do not comply with its expectations. A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions. Students whose programs are identified as containing significant overlaps will have substantial mark penalties applied, or be referred to the Student Center for possible disciplinary action.

Nor should you post your code to any public location (`github`, `codeshare.io`, etc) while the assignment is active or prior to the release of the assignment marks.