CMPS 101 - Programming assignment 1 - W13
Implement the Game of Life

Handed out 1-10-13 **Due 1-22-13 in locker**

In this assignment, you will implement a mathematical game known as "Game of Life" devised by John Conway in 1970. This zero-player game is set on a two dimensional array of cells (potentially infinite in dimensions). Each cell is either alive or dead. The game starts with an arbitrary pattern of cells set to live status. The rules of game are as follows.

If an alive cell has two or three alive neighboring cells, it stays alive.

If a dead cell has exactly three alive neighboring cells, it comes to life.

Otherwise, the cell status remains unchanged. A live cell stays alive and dead cell stays dead.

A neighboring cell is any of the eight cells adjacent or diagonally adjacent to the given cell.

These carefully chosen rules allow for very interesting evolution of cells starting with even simple patterns. You can see the game in action at the following website, "http://www.conwaylife.com".

The life is staged on a two dimensional grid, with each cell at unique (x, y) coordinates. The top left corner has coordinates (0,0) and x increases as one moves left and y increases as one moves down.(Typical choice for graphics devices). Simplest choice for representing the cells would be to use a two dimensional integer array, where each cell contains a simple integer with value=1 for live cell and 0 for dead cell. (The cells may also contain other information like neighbor count etc). Note that this simple structure will not allow a O(n) update algorithm required by the assignment. To perform updates efficiently, an additional array(unInit-array) can be used (There are many other algorithms, but this is what you will need to implement for this assignment). The unInit-array keeps track of only the live cells. An entry in the unInit-array points to the corresponding entry in the grid and vice-versa. This reference loop serves as a way to verify that the entries are valid. (Note that you will not need to initialize either of the arrays). The figure 1 shows the relationship between the arrays. Attributes like status of a cell, count etc are not shown in the figure, but are necessary for your program. These attributes can be kept in the unInit array rather than grid to save space.

Basic operations on this unInit array are as follows.

Insertion: A new element is always inserted at the end and this entry and the corresponding cell in grid are updated to refer each other.

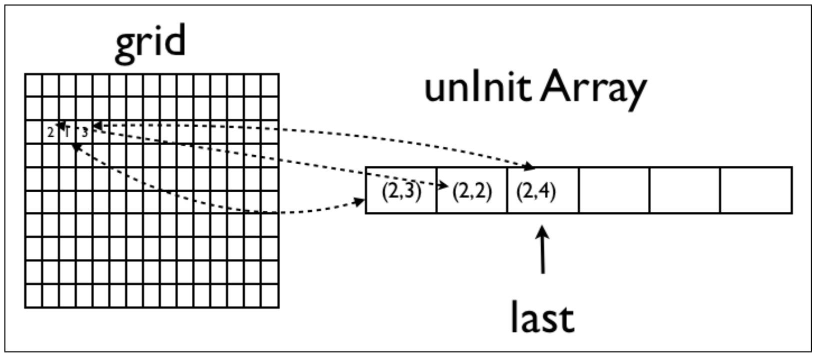

Figure 1. Representation of a blinker object

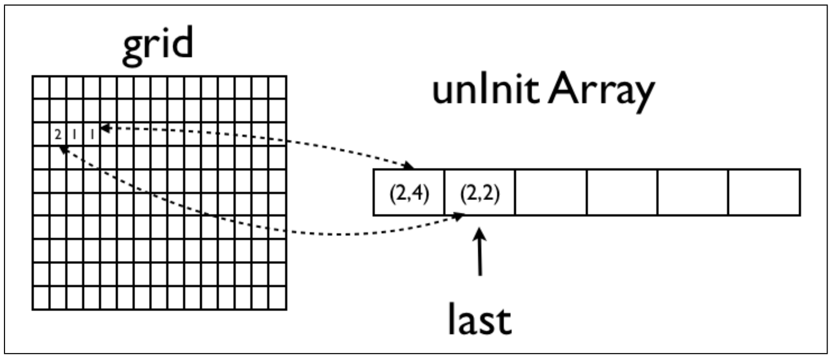

Figure 2. state after deleting cell (2,3) from unInit Array

Deletion: Just deleting an element will not work as it leaves a hole in the array. Fortunately it is simple to fill this hole by swapping in the last array element in to this just vacated entry. (We can do this because the order of entries is not important for our purposes). When the swap is done, it is important to ensure that the reference from the grid to the swapped entry is still correct. As an example, deleting cell (2,3) from state shown in figure 1 leads to the figure 2. Note that grid entry (2,3) may still refer to its location in unInit array (index 1 in this case). However since the element at index 1 in unInit array does not point back to the deleted entry, the grid entry is invalid.

Various phases involved in the update are as follows. These phases are primarily for descriptive purposes and you may choose to combine them for efficiency or convenience.

Adding neighbors: In this phase, the number of neighbors for each cell will be counted. This can be done in O(n) by walking through the aux-array. For each cell, there are eight neighbors. (If the cell has coordinates (x,y), then (x-1,y-1), (x, y-1), (x+1,y-1), (x-1,y), (x+1,y), (x-1,y+1), (x, y+1), (x+1,y+1) are the neighbors.) Add the neighbors that are not already in the aux-array. (How do you check if a given cell is in the unInit array?). It could be easier to add the neighbors to a different array, i.e. one array for live cells and one array for their neighbors. Since the game rules for how alive cell propagate are different than rule for dead cells, this division of cells into different arrays can make things simpler.

Counting live neighbors: The previous stage has gathered all the potential cells into the aux-array. (Note that no other cell needs to be updated and thus a lot of computation is saved as we don't need to scan through entire grid.). It is easy to go through this list and count number of live cells adjacent to each cell. This live neighbor count is what is needed to decide which cell will propagate to the next generation.

Cell propagation: Once the neighbor counts are known, the status of cells is determined and cells that will stay alive in the next generation are updated. (Using the rules of Life). You can perform this update within same aux-array (by deleting and inserting as necessary) or you can add propagating cells into a temporary array and then replace the unInit-array.

Display: The cells are drawn onto a Graphical device or to a terminal device. This phase will be kept simple so that programming can focus on the main aspects of the game.

Additional details and advice

Your program should read input from a file. The input file contains how many generations of life you need to simulate as well as the coordinates of the initial pattern. We will provide some examples for you to work with, and you should try to test with more patterns of your own. The input file format is as follows. Only the numbers will be in the file. The comments following '<—' below are only explanatory.

```
10     <--- number of generations that you will need to simulate
6      <--- number of cells in the object about to be read
30 19  <--- (x, y) coords for the various cells of the object
20 20
21 20
22 20
30 20
30 21
```

Make your algorithm as simple as possible so that you have less of a chance to introduce bugs.

We suggest the following outline:

Implement read/write procedures. Read in your creature from a file and then print it out the grid.

Test your insert and delete procedures for the unInit array. If these work correctly, most of the work is done. Modular design will help here.

Check that you are able to correctly add the cell neighbors. Only the ones not already in the array will need to be added. While adding the neighbors, it is easy to compute the counts too. print out the unInit array to ensure this works correctly.

Now apply the rules correctly and perform update. If you chose to update within the same unInit array, remember that you need to scan backwards as delete procedure moves element at the last to the deleted slot.

Test your procedure on some small creature. The blinker object is made of just 3 cells and is very useful for testing.

Test it on some larger creature that we will provide.

As far as printing/displaying is concerned, print a 30x30 board. If a creature wonders off then ignore it.

All programs should give a half a page outline of the algorithm you used. In particular you need to justify that your algorithm is O(n). Please do start early and discuss any doubt/issues in advance with the TA and Professor.

What to submit

Submit your code for game of life, programmed in a language of your choice. The code documentation should give a half page outline of the algorithm used and justification why the algorithm is O(n). Note in particular how the data structure you used helps you in improving run time. The code needs to be well documented so that grader can get a good idea of what each of your procedures do.

Program modularity, clarity, documentation etc along with correct implementation will be considered for grading. The grading breakdown is as follows. 60% correctness, 20% modularity, 20 % documentation. Submit instructions will be updated here soon.

Extra credit of 20%

Implement the program on a mobile platform.

Implement a print routine which will print all live objects.