CMPS 101 - Programming assignment 2 - S11
Implement the Game of Life
Handed out 4-14-11 Due 4-26-11 in locker

In this assignment, you will implement a mathematical game known as "Game of Life" devised by John Conway in 1970. The 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 (read from input) 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 neighbouring 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".

A Simple implementation of the game works as follows.

Representation 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. Simplest choice for representing the cells would be to use a two dimensional integer array, where each cell is a simple integer with value=1 for live cell and 0 for dead cell.

Counting cell neighbors In this phase, the number of neighbors for each cell will be counted. Simple, but inefficient implementation would be to iterate throught the 2-d array and count neighbors for each cell. The cell state does not change during this phase. Note that this method will count dead cells too and hence wastes computation.

To be efficient, this operation must be linear in the order of number of live cells.

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. It is necessary to keep another array to update the cell state. The counting and cell propagation are performed together, i.e, count a cell and if it should be alive, then update the cell in the next 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.

Your task is to implement the game of life using the specifications given below. The input will be as follows. (The y coordinates increase downwards and x increases to the left. The top left corner is 0,0. Ofcourse you can translate the data to your convenience). gen: number of generations to simulate. n: number of cells in input pattern coordinates: x1 y1 ..... xn yn

Sample input is as follows.

```
10
5
51 48
52 49
50 50
51 50
52 50
```

Display this in a 100x100 grid after the given number of generations. You will need to initialize the cell with the pattern given in input and then simulate the game. Next generation of the life is computed from the current state using the rules given above.

Life with pointers

Representation: Each cell is represented as a tuple (x,y, state,count, next), where state=live or dead and next is index of the next live cell in the row. All the live cells are inserted in sorted row major order into a singly linked list, which will be a sparse representation of the life grid.

Count & Propagate: The counting should be linear in order of live cells. Any of the linear order algorithms described in the class can be used for this. Note that you need to be careful at the boundaries. Your program should give a clear description (half page) of the algorithm you are using and justification that it is O(n), where n is number of live cells.

Display: The printing function is very simple. You will print out the cell grid to the terminal. For a dead cell you will output a blank ' ' and for a live cell you will output a 'x'. You can also output the total number of live cells in the grid. The output for each generation is followed by a dividing line made up of hyphens of suitable length. '—— ..... —-'. If you can keep track of range of x and y coordinates between which live cells occur, you can only display that particular area to simplify your output. If you want a simpler option, just print a 50x50 board. You will also need to implement an option for print whereby it will display the neighbor counts of the cells in the display area.

Counting algorithm 1 : using 8 linked lists

This algorithm maintains 8 lists, one for each adjacent contributor of a cell. (name them W, NW, N so on). The algorithm traverses linked list of live cells and at each cell adds an element to the respective list. For example, into North list we add the element x, y-1, state=0, count=1, into W list we add the element x-1, y, state=0, count=1 so on. After processing all the live cells, we need to process these 8 lists. The processing involves merging lists to remove the duplicate coordinates and updating the correct counts (This operation is similar to the merge operation in the merge sort algorithm). The elements with wrong counts need to be deleted. It is possible to reduce the number of lists used to 3.

Counting algorithm 2 : 3 finger scan

Instead of maintaining several lists, you can ensure that each required element is inserted exactly once. Three pointers are used to scan a particular row, one pointing to previous row, one to the current row and other the next row. We need three pointers because we are dealing with singly linked list representation. (if we used 2d arrays, this would be not needed). If cell (x,y) is being processed, we need to way to know about any cells in the top rows to determine count of cells in the current row. Same thing with the next row. Additional details on this algorithm will be provided shortly.

Keep track of the number of live cells in your game along with other stats of your interest. As before, record any interesting observations in the README file.

Assignment Schedule

Try to develop the project in stages if it seems a bit hard. 1. Start with code that reads the input. 2. develop the display code. 3. Develop the counting algorithm and carefully test it. Convince yourself that you have a linear order algorithm. Try with just one step using simple patterns. Try to keep the algorithm as simple as possible. 4. Now pull in all the code together and use it study the game of life. Observe the behaviour with various patterns. See how even seemingly simple patterns show explosive growth, while other patterns simply disappear.

Extra Credit

The extra credit of 20 points will be awarded for using graphics to show the evolution of the game. Or design a printing routine that prints all live cells even if they wander off the page. Reason that you routine is O(n) again.

What to submit

Submit your code for the implementation. You will also need to provide a makefile to compile the program and submit it. (Not submitting the makefile will incur significant penalty). The makefile when invoked should generate a executable or jar file (depending on your language), the name of which should be 'lifeSim' (with an extension if needed). Program modularity, clarity, documentation etc along with correct implementation will be considered for grading. The grading breakdown is as follows. 80% correctness, 10% modularity, 10 % documentation. Submit instructions are as follows.

command : submit cmps101-mw.s11 prog2 [filenames]

The submit window will close exactly at mid night of the deadline date. No late submissions will be accepted.