**CS4306 Algorithm Analysis**

Fall 2021

Department of Computer Science

Kennesaw State University

**Programming Assignment 1: Game of Life**

**Due Date: Thursday, September 2, 2021 (by 11:59pm)**

**Problem Description**:

Life is played on a grid of square cells. A cell can be **live** or **dead**. A live cell is shown by putting a marker (#) on its square. A dead cell is shown by leaving the square empty. Each cell in the grid has a neighborhood consisting of the eight cells in every direction including diagonals. Cells on the four edges of the grid have fewer than eight neighbors, of course.

*Rules of the game*: To apply one step of the rules, you have to count the number of live neighbors for each cell. What happens next depends on this number.

* A dead cell with exactly three live neighbors becomes a live cell (birth).
* A live cell with two or three live neighbors stays alive (survival).
* In all other cases, a cell dies or remains dead (overcrowding or loneliness).

**Note:** The number of live neighbors is always based on the cells **before** the rule was applied. In other words, you must first find all of the cells that change before changing any of them.

**Data Structure**: Two-dimensional array/ Vector

**Sample Input**:

The program can be run on a 5x5 and 10x10 array for simplicity. You should simulate upto the 50th generation (10, 20, …, 50). The input should contain the initial values of the cells.

0 0 0 0 0

0 1 0 0 0

0 0 1 0 0

0 0 0 1 0

0 0 0 0 0

**Sample output**:

The output should be displayed in text mode.

The system time should be output in msec.

**Report**:

Give a brief description of the analysis of the problem. This should include time and space complexity. A graphical representation of the variation of time with generation should be provided. *The report should be no more than 3 pages*.

**Language to be used**: C/ C++/ Java

*Report*

*My Anh Huynh*

1. Description of the analysis of the problems:
2. The game of life rule:

* Birth: a dead cell with exactly three live neighbours becomes a live cell
* Survival: a live cell with two or three neighbours stays alive
* Overcrowding/ Loneliness: in all other cases a cell dies or remains dead:



1. Data structure, time and complexity:

Generally, with different types of representation of matrix or two dimensional array have different performance in time and space cost.

Use a two dimensional array 10x10 that generated randomly.

The problem uses nested loops contain sizes M and N so the cost is O(M\*N) to generate the original array and update cells.

For example:

// populating the grid  
for (int i = 0; i < M; i++) {  
  
 for (int j = 0; j < N; j++) {  
 int randomNum = randomNumberGenerator.ints(0,2).findFirst().getAsInt();  
 grid[i][j] = randomNum;  
 System.*out*.print(grid[i][j] + " ");  
 if (j == M - 1) {  
 System.*out*.println();  
 }  
 }  
 }

1. System time:

Also, we use the function System.nanoTime() to calculate the time that the program run for each generation execute.

Graphical user interface, text

Description automatically generated

From every execute time, we collect the time and draw the figure below:

Table

Description automatically generated

Figure 1: System Time

1. Source code and screen shot:

package package1;  
  
import java.util.Random;  
  
public class Solution {  
 public static void main(String[] args) {  
  
 int M = 10;  
 int N = 10;  
 Random randomNumberGenerator = new Random();  
  
 // two dimensional array of int value  
 System.*out*.println("The original board");  
 int[][] grid = new int[M][N];  
 // populating the grid  
 for (int i = 0; i < M; i++) {  
  
 for (int j = 0; j < N; j++) {  
 int randomNum = randomNumberGenerator.ints(0,2).findFirst().getAsInt();  
 grid[i][j] = randomNum;  
 System.*out*.print(grid[i][j] + " ");  
 if (j == M - 1) {  
 System.*out*.println();  
 }  
 }  
 }  
 //display the grid  
 System.*out*.println("Original generation");  
 for (int h = 0; h < M; h++) {  
 for (int q = 0; q < N; q++) {  
 if (grid[h][q] == 0) {  
 System.*out*.print("[]");  
 } else System.*out*.print("#");  
 if (q == M-1)  
 System.*out*.println();  
  
 }  
 System.*out*.println();  
 *nextGeneration*(grid, M, N);  
  
 // function to generate the next generation  
 }  
 }  
  
  
 public static void nextGeneration ( int grid[][], int M, int N){  
 int[][] future = new int[M][N];  
 // loop through every cell  
 for (int l = 1; l < M - 1; l++) {  
 for (int m = 1; m < N - 1; m++) {  
 // to find out the number of alive cell  
 int aliveNeighbours = 0;  
 for (int i = -1; i <= 1; i++) {  
 for (int j = -1; j <= 1; j++) {  
 aliveNeighbours += grid[l + i][m + i];  
 /\* the cell needs to be subtract from  
 its neighbours as it was counted before  
 \*/  
 aliveNeighbours -= grid[l][m];  
 /\* the rule life are implement here  
 cell is alone and dies  
 \*/  
 if ((grid[l][m] == 1) && (aliveNeighbours < 2)) {  
 future[l][m] = 0;  
 }  
 // cell dies due to over population  
  
 else if ((grid[l][m] == 1) && (aliveNeighbours > 3))  
 future[l][m] = 0;  
 // a new cell is born  
 else if ((grid[l][m] == 0) && (aliveNeighbours == 3))  
 future[l][m] = 1;  
 // remain the same  
 else  
 future[l][m] = grid[l][m];  
  
  
 }  
  
 }  
 }  
  
 }  
 //up date the cells  
 System.*out*.println("Update cells");  
 for (int i = 0; i < M; i++) {  
 for (int j = 0; j < N; j++) {  
 if (future[i][j] == 0)  
 System.*out*.print("[]");  
 else  
 System.*out*.print("#");  
 }  
 System.*out*.println();  
 }  
  
 long startTime = System.*nanoTime*();  
 long endTime = System.*nanoTime*();  
 double duration = (double)(endTime- startTime)/1000000;// divide by 1000000 to get millisecond  
 System.*out*.println("duration is:"+duration);  
 }  
 }

A picture containing text, electronics, keyboard

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated with low confidence