Python Implementation of Conway's Game of Life

Brad Green & Edwin Weill

November 29, 2016

1 Introduction

The overall idea behind this project is to design a system that simulates *Conway's Game of Life*, a cellular automaton. Rather than being a "game" in which players interact with each other, this particular "game" is a zero-player game meaning that it goes through evolutions without any interaction with a player. The only interaction the player has with the system is configuring the initial state and then lets the system evolve over time.

The game is played on a two-dimensional grid of squares called *cells*. Each cell is categorized as either *alive* or *dead* at a particular "generation" of the simulation. Each subsequent generation is determined by the interaction between neighbors (each cell interacts with its eight surrounding neighbors). There are many rule variations that change the behavior of the system (i.e. how cells are spawned and how cells die). The following set of rules is based on a rule variation known as B3/S23 (born with 3 neighbors, survive with 2 or 3 neighbors):

- **Rule 1**: A living cell with fewer than 2 living neighbors dies (under-population)
- **Rule 2**: A living cell with 2 or 3 living neighbors lives to next generation
- **Rule 3**: A living cell with more than 3 living neighbors dies (over-population)
- **Rule 4**: A dead cell with 3 neighbors becomes alive (reproduction)

2 Literature Review

3 Implementation

The initial implementation simply uses the rules described above (or a similar rule strategy) to determine each generation in the evolution of the system and makes an updates based on each cell's neighbors. The implementation iterates through every cell in the grid (alive or dead) and determines its next state based on the rules and neighbors. The psuedocode in Algorithm 1 illustrates this "naïve" algorithm implementation:

4 Optimizations

The above implementation is rather simplistic and therefore it does not perform well for larger board sizes. The naive implementation is of computational order O(width*height) which means that every cell must be visited during each evolution between generations. This is highly inefficient when the board size gets large, therefore slowing down its execution. For this reason, optimizations

Algorithm 1 Naïve Implementation Conway's Game of Life

```
1: procedure NAIVE CONWAY(width, height, cell_initialization, max_iterations)
2:     grid ← initialize_grid(cell_initialization)
3:     count ← 0
4:     while count < max_iterations do                ▷ Update until limit reached
5:         if no change from previous generation then
6:             exit
7:         count ← count + 1
8:         for y < grid_height do
9:             for x < grid_width do
10:                if cell is living then
11:                    if cell has 2 or 3 neighbors then
12:                        cell.state ← dead
13:                else
14:                    if cell has 3 neighbors then
15:                        cell.state ← alive
```

need to be employed utilizing various data structures to alleviate the need to "visit" every cell in the grid.

Set of Living Cells

The first optimization technique makes use of an additional set which holds all cells which are currently living. The strategy only checks for modifications in cells that are currently living and all neighbors of the living cells. For each cell, we keep a count of how many cells it touches, which can be used to check the rules and add cells to the new *living* list. This implementation is much more efficient as the number of rule checks is significantly decreased for most instances of a typical grid layout, O(number of living cells).

Quadtree

ADD PARAGRAPH ABOUT QUADTREE HERE

5 Results

5.1 Visualization Results

There are many different patterns that the game of life lends itself to when started in certain configurations. For example, there are some patterns which do not change not matter how long the simulation is run (unless they are interacted with by another living cell). There are others that oscillate between a set number of states; these patterns are called oscillators. Figure 1 illustrates a *box* which is an example of a still pattern while Figure 2 illustrates a pattern with a period (an oscillator).

There are many other types of patterns that facilitate good visualizations including glider guns (shown in Figure 3). The glider is created from the *gun* every 30 iterations based on the B3/S23 rule strategy. Another interesting example is a single live cell beginning the game with the automaton B1/S12 resulting in a very close approximation of the Sierpinski triangle. Figure 4 illustrates this example.

Figure 1: Still Positions 1 & 2

Figure 2: Oscillator Positions 1 & 2

Figure 3: Glider Gun Positions 1 & 2

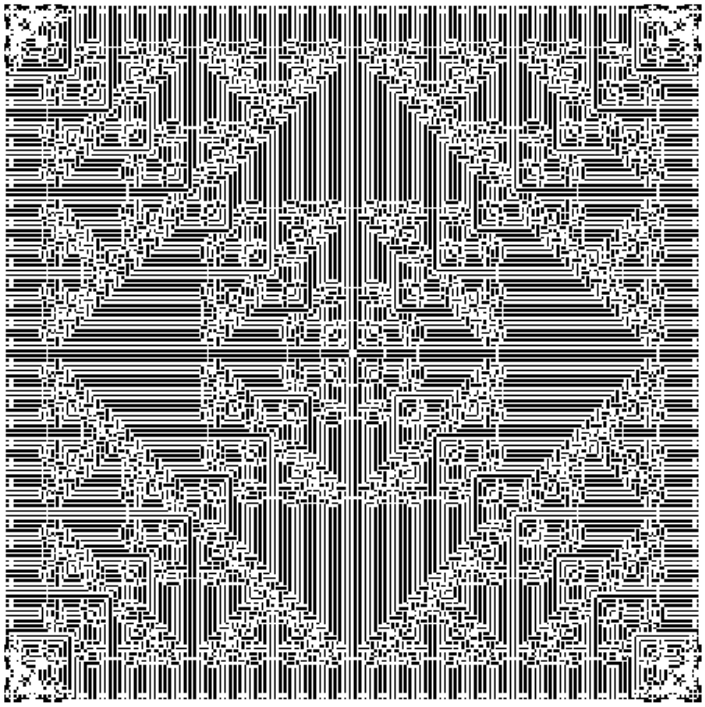

Figure 4: Sierpinski Triangle

5.2 Performance Results

6 Discussion

7 Distribution of Work