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*[1], a cellular automaton. The Game of Life is atypical, because this particular "game" is a zero-player game. This means that it "plays" through evolutions without any interaction with a player. The only interaction the player has with the system is configuring the initial state and then allowing 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 Implementation

The initial implementation simply uses the rules described above (or a similar rule strategy) to determine each generation as the evolution of the system and makes 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 the state of their neighbors. The psuedocode in Algorithm 1 illustrates this "naïve" algorithm implementation:

3 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 leading to much longer execution times. For this reason, optimizations need to be employed utilizing various data structures to alleviate the need to "visit" every cell in the grid.

Algorithm 1 Naïve Implementation Conway's Game of Life

```
procedure NAIVE CONWAY($width, height, cell\_initialization, max\_iterations$)
    $grid \leftarrow initialize\_grid(cell\_initialization)$
    $count \leftarrow 0$
    while $count < max\_iterations$ do                ▷ Update until limit reached
        if $no\ change\ from\ previous\ generation$ then
            exit
        $count \leftarrow count + 1$
        for $y < grid\_height$ do
            for $x < grid\_width$ do
                if $cell\ is\ living$ then
                    if $cell\ has\ 2\ or\ 3\ neighbors$ then
                        $cell.state \leftarrow dead$
                else
                    if $cell\ has\ 3\ neighbors$ then
                        $cell.state \leftarrow alive$
```

Set of Living Cells

The first optimization technique makes use of an additional set which holds all currently living cells. The strategy only checks these cells and all 8 of their neighbors to evolve to the next generation. 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, leading to O(number of living cells) complexity. As sets are implemented by hash tables in python which have O(1) access times, the overall complexity is O(number of living cells).

Quadtree

The next optimization made use of a QuadTree data structure to save space. A QuadTree[2] is a modification of a typical tree structure that requires each node have four children or be a leaf node. This specification allows the structure to naturally represent spatial data while requiring very little memory as only existing elements are stored in the tree, as opposed to an array that must be large enough for all elements. We use the QuadTree to represent the structure of living cells, meaning every leaf node of the structure represents a living cell in the Game of Life grid. Sets are used, much like in the first optimization, to maintain performance while saving space during evolution. As this is a tree structure, the depth of any leaf is O(log(n)) and inserting nodes takes O(log(n)) time. This leads to worst case performance of O(log(n) * living cells) Figure 1 shows a representation of a Game of Life grid stored in a QuadTree.

4 Results

4.1 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 from one generation to the next, called still lifes. There are others that oscillate between a set number of states; these

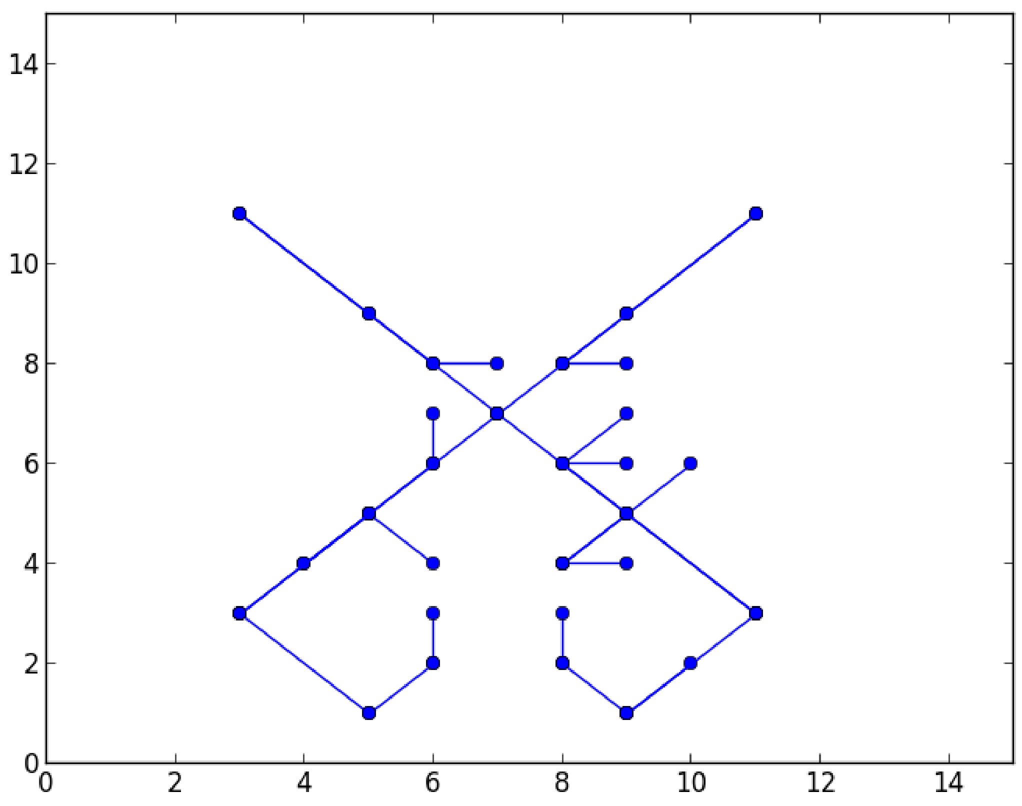

Figure 1: QuadTree Representation of Lightweight Spaceship

patterns are called oscillators. Figure 2 illustrates a *box* which is an example of a still life while Figure 3 illustrates a pattern with a period (an oscillator).

Figure 2: Still Life Positions 1 & 2

Figure 3: Oscillator Positions 1 & 2

There are many other types of patterns that facilitate good visualizations including glider guns (shown in Figure 4). 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[3]. Figure 5 illustrates this example.

4.2 Performance Results

To compare the methods, we created a Game of Life grid with set initial conditions (each cell has a 30% chance to start as alive) to verify all methods completed the same evolutions. 100 generations were simulated for each method using the B3/S23 ruleset. Figure 6 shows the memory taken by the different optimization methods for growing grid sizes. Figure 7 shows the time taken by each method for the same test conditions.

Figure 4: Glider Gun Positions 1 & 2

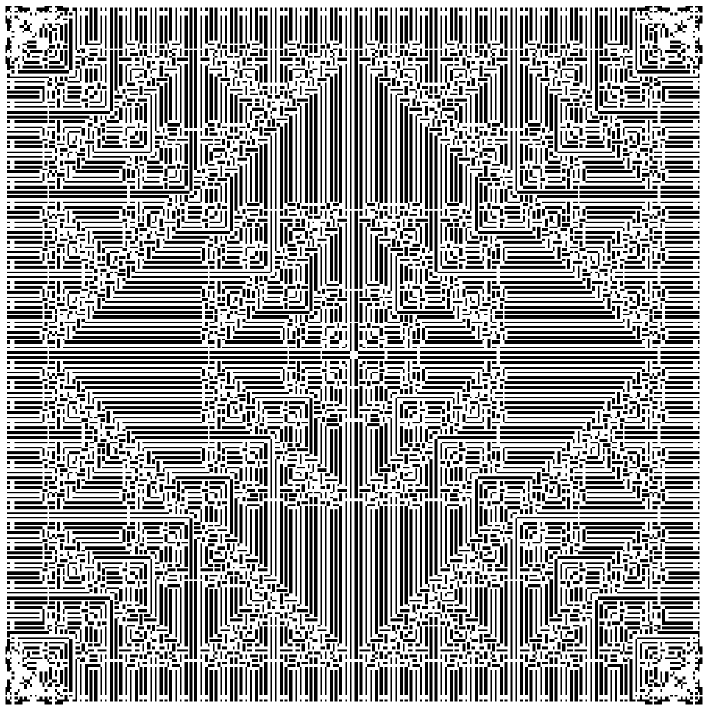

Figure 5: Sierpinski Triangle

5 Discussion

As shown in Figure 6, the QuadTree implementation used less memory than both the Naive and Optimized methods as only alive nodes are saved. The graph represents the memory usage of the

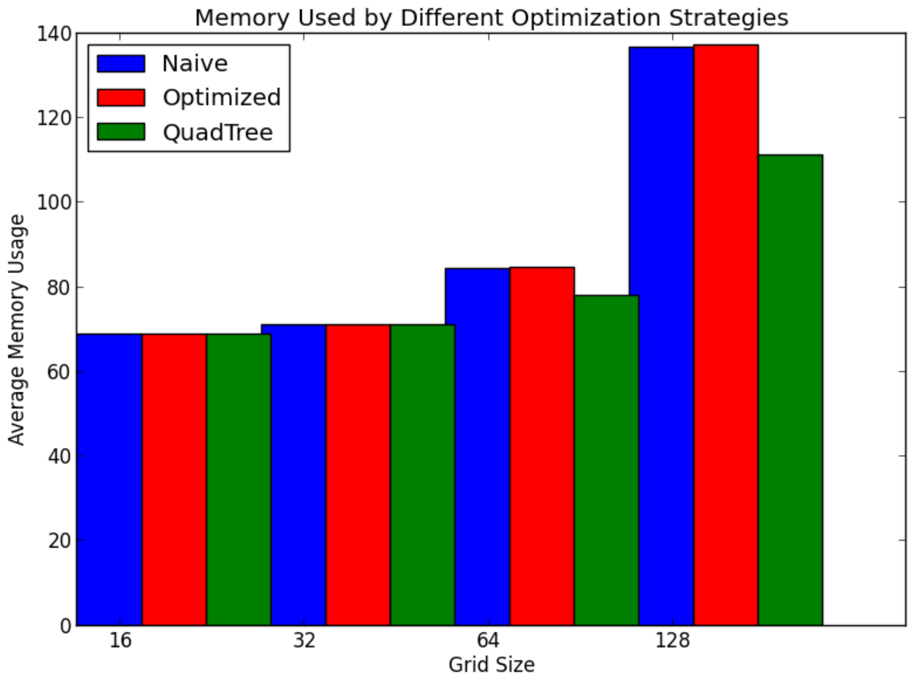

Figure 6

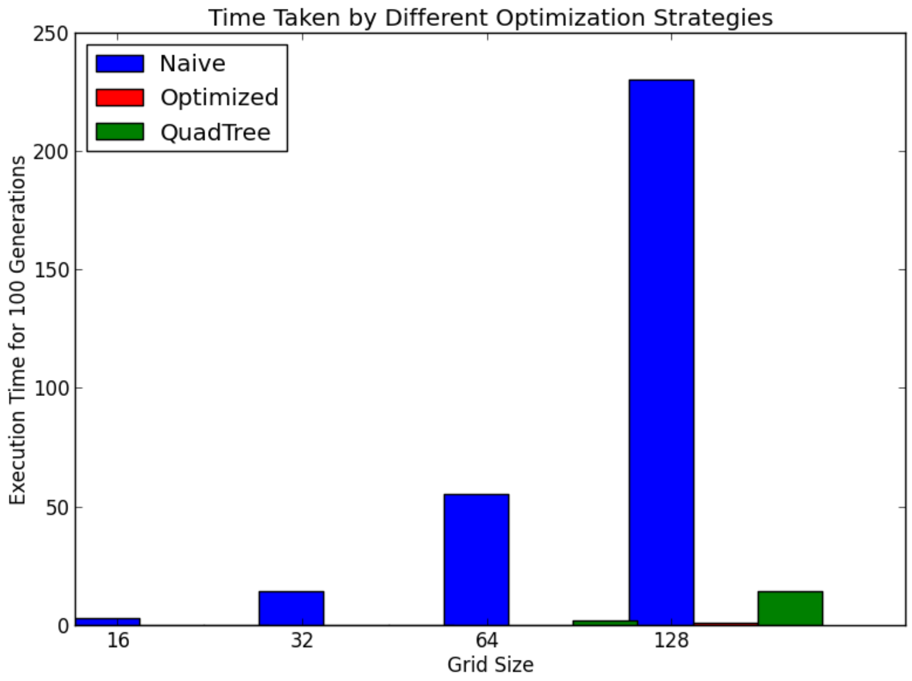

Figure 7

entire python program, including all overhead for imported modules. If we take the 16x16 grid as a baseline and normalize the 128x128 grid to this, we can see the QuadTree implementation saves significant memory over the other methods to store the Game of Life grid. Further, the tested version of the QuadTree does not *prune* empty branches to save on time, so there is an added overhead of empty branches stored in the tree. Figure 7 shows that our optimized version runs fastest due to the constant access times of the hash tables for the sets. If we were to prune the QuadTree, it would take significantly longer to run (pruning recursively checks all branches and deletes any which have no leaves).

6 Distribution of Work

Brad wrote the backend code to run the Game of Life for naive, optimized, and QuadTree methods. Brad also completed the profiling for memory and timing results. Eddie completed the Unit Testing and implemented all of the graphical results. Eddie polished the Game of Life top-level code to include command line arguments and prepped the scripts for presentation. The presentation and report was created by Eddie and each partner filled in the sections corresponding to their work completed.

References

[1] Conway's Game of Life Wiki: *https://en.wikipedia.org/wiki/Conwayś_Game_of_Life.*

[2] Quadtree: *http://www.fundza.com/algorithmic/quadtree/index.html.*

[3] One Cell Thick Pattern: *http://www.conwaylife.com/w/index.php?title=One_cell_thick_pattern.*