Conway's Game of Life

Suran Warnakulasooriya

Contents

1 The Origins of Life

The road to the Game of Life begins with the Turing Machine, a machine that operates on an infinitely long strip of tape divided into "cells". A "head" lies atop a cell and reads the symbol on it. Depending on the symbol and a predetermined set of instructions, the head first writes a symbol, moves the tape left or right, and then continues or stops depending on the current symbol and the machine's current state. Much like the Game of Life, despite having simple rules, a Turing Machine is deceptively complex, capable of replicating the logic of any computer algorithm. John von Neumann used the Turing Machine in his definition of life, in which he described a being that can reproduce itself and simulate a Turing Machine. Neumann was looking for an engineering solution that utilized electromagnetic components. His ideas could not be rendered into reality using the technology of his time. However, Stanislaw Ulam offered to simulate Neumann's theoretical machine using the first cellular automata. The cellular automata acted much like a Turing machine, but instead sweeping across a 2 dimentional tape.

Both the Turing Machine and cellular automaton operated on predetermined instructions or rules. Changing these rules would change how the machines would react to the observed cell. Both machines opened new questions in mathematical logic and proposed the possibility of simulating them. In 1968 at the University of Cambridge, John Conway began experimenting with combinations of cellular automaton rules in the hopes of producing an unpredictable cellular automaton. A purely unpredictable machine is impossible since the nature of a cellular automaton makes it inherently deterministic. If given the rules and the state of the tape, a person could predict what the state of the tape will be after any number of iterations. Conway had a different, more chaotic definition of unpredictability. With the same set of rules, he wanted slightly different starting configurations to produce vastly different results. The ruleset had to prevent explosive growth, allow small patterns with unpredictable growth, and allow for the creation of a Von Neumann universal constructor, which is a self-replicating machine. Conway wanted a simple rulset that met this criteria.

2 3 Simple Rules

Conway ultimately arrived at the following rules for a cellular automata that met his criteria:
Rule 1: Any live cell with two or three live neighbors survives.
Rule 2: Any dead cell with exactly three live neighbors becomes alive by reproduction.
Rule 3: Any live cell with less than two live neighbors dies by underpopulation.
Rule 4: Any live cell with more than three live neighbors dies by overpopulation.

These 4 rules can be condensed into the following 3:
Rule 1: Any live cell with two or three live neighbors survives.
Rule 2: Any dead cell with three live neighbors becomes a live cell.
Rule 3: All cells that do not meet the above criteria are left unchanged.

The state of every cell updates simultaneously according to the above rules. All cells update simultaneously since if they did not, the order at which cells are updated would affect the fate of cells that have not yet been updated. Each update is a new generation, sometimes called an iteration or a tick. To evaluate a cell is to determine its next state without updating it. The neighbors of a cell are the 8 cells that form its immediate perimeter. Some have experimented with different game boards entirely. The regular Game of Life is played on a 2d array of square or occationaly rectangular cells (which is an aesthetical and not a mechanical change). However, the game can be played with triangular or hexagonal cells. A triangular cell has 6 neighbors while a hexagonal has 12. Alternatively, the game of life could be played in 3 dimensions, the cubed version of which would have 14 neighbors per cell. Changes in neighbor count affect the probability that a cell can become alive, meaning that a pattern will likely not have the same outcome in different versions of the Game of Life. These variations are formally called Life-like Cellular Automata. If one wishes to stick to the regular 2d array of squares, they could consider the opposite sides of a finite playfield to loop around, resulting in a Game of Life played on a torus.

3 More Than a Game

Conway's Game of Life first appeared in Martin Gardner's "Mathematical Games" column in the October 1970 issue of Scientific American. The Game of Life is, as stated previously, an incarnation of the Turing Machine. As such, it can mimic any algorithm. This facet of the Game of Life has kept the attention of computer scientists and mathematicians. The game developed a cult following of students and professors discovering new patterns that persist to this day. Besides mathematicians, biologists, economists, and philosophers have also been intrigued by the Game of Life due to its ability to produce emergent and self-organizing systems. The Game of Life is a zero player game, with its gameplay being dictated by the predetermined rules and cell configuration laid before it. Despite the lack of an active player, the Game of Life allows for organization to spontaneously emerge. Biologists have likened the emergence of complex structures in the Game of Life to the evolution of life on earth. Philosophers have gone further to connect it with consciousness and free will, all of which in this analogy are illusions produced from the inherently deterministic rules of the Game of Life and real world laws of physics. After all, Conway created the Game of Life to produce the illusion of chaos. In the words of Martin Gardner, "Because of its analogies with the rise, fall and alternations of a society of living organisms, it belongs to a growing class of what are called 'simulation games'–games that resemble real-life processes (Scientific American Oct 1970)". The graphical and computational ease of the Game of Life allows it to run endlessly on almost any system. This accessibility has allowed communities to grow and create many patterns of varrying types as shown in the next chapter. Some have even gone as far as to emulate a computer within the Game of Life or even the Game of Life itself.

4 Many Species

4.1 Still Lifes

The fact that a live cell can remain alive under the condition that it has 2-3 live neighbors is utilized in the following configurations. Contrary to the goal of the Game of Life's creation, these configurations are "Still Lifes" and do not change from generation to generation. The reader can verify for themselves that all of these patterns are static.

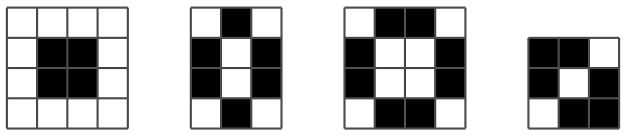

Figure 1: The "Block", "Beehive", "Pono", and "Ship" are still life configuations.

All of the configurations in 1 do not change with progressing generations. Each live cell satisfies the survival rule and none of the dead cells satisfy the reproduction rule. Consequently, no live cells die and no dead cells come alive, leaving a pattern that does not change over generations.

4.2 Oscillators

Other patterns are built to oscillate between a finite amount of states before returning to the first state. The number of states is the "period" of the oscillator. For example, the "Blinker", "Toad", and "Beacon" shown below are period 2 oscillators that go through 2 states before returning to the first (including the beginning state). The only criteria for a pattern to be an oscillator is to eventually reach a pattern identical to how it started. Naturally, oscillators loop through the same set of states indefinitely. The reader can verify for themselves that all of these patterns oscillate. Figures 2, 3, and 4 are period 2 oscillators.

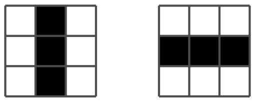

Figure 2: The "Blinker", which alternates between the above two states indefinitely.

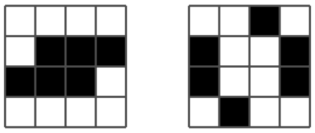

Figure 3: The "Toad".

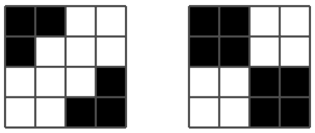

Figure 4: The "Beacon".

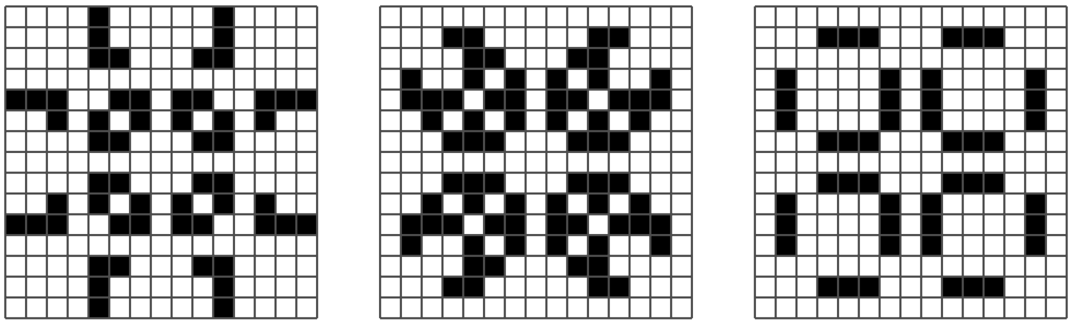

Figure 5: The "Pulsar" of period 3.

4.3 Spaceships

Spaceships (see figures 6 and 7) are cousins of the ocillators. They cycle through a finite number of states, but shift their position with each cycle. The orientation of the pattern will determine the direction a spaceship moves in. Spaceships are often produced by other complex patterns like the Glider Gun in figure 8.

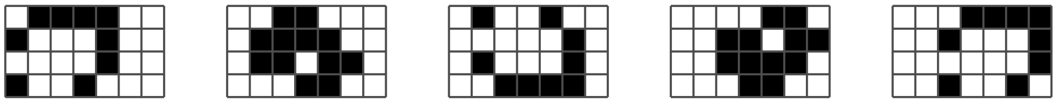

Figure 6: The "Light Weight Spaceship (LWSS)" of period 4, which moves two cells to the right after each cycle.

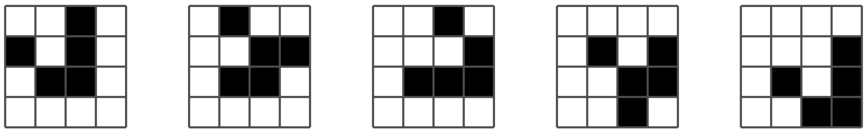

Figure 7: The "Glider", a period 4 spaceship which moves right and down 1 cell after each cycle.

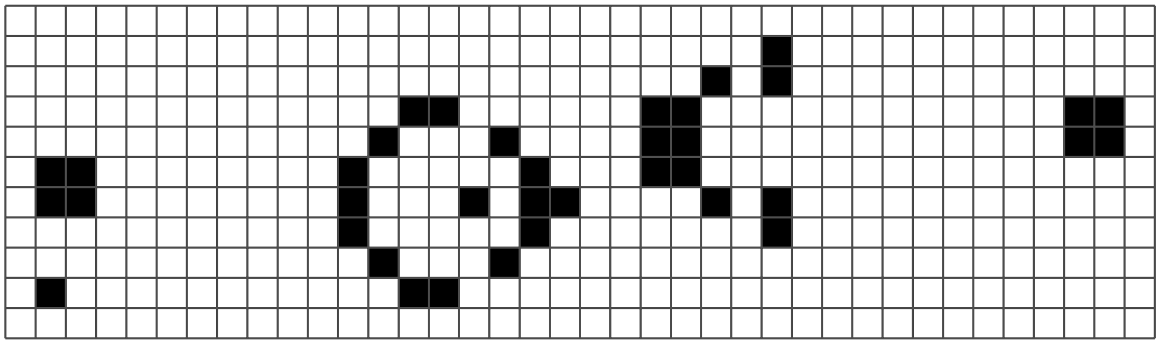

Figure 8: The "Gosper Glider Gun" is a nonending pattern that produces a glider on the 15th tick and then once every 30 ticks.

4.4 Methuselahs

When experimenting with random configurations, one will notice that almost all patterns will either die out or stabilize into a collection of still lifes and oscillators. Of course, some patterns last longer than others before reaching such a state, the long living of which are called Methuselahs (figures 9, 10, and 11). Methuselahs and all other patterns are considered "chaotic" until they either die out or stabilize.

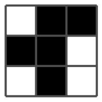
Figure 9: The "R-Pentomino", the first discovered methuselah.

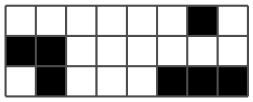
Figure 10: The "Diehard".

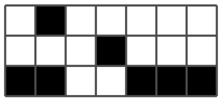
Figure 11: The "Acorn".

5 The Complexity and Limits of the Game of Life

Methuselahs simply last long before stabilizing, but some patterns have been found to properly last forever. These patterns often act like spaceships but leave a trail of debris behind, causing the number of live cells to grow infinitely. The glider shown in 7 appears naturally from many patterns and is often used in patterns that attempt to emulate a computer in the Game of Life, where the presence or absence of a glider corresponds to a 1 or a 0. Additionally, gliders can be used to create logic gates. Some have discovered that two gliders that collide at the right angle will destroy each other, allowing clever players to create large arrays built from rows of gliders that destroy each other at the end. These arrays can be shaped in squares to recreate the Game of Life within itself. Gliders interact with other patterns in interesting ways as well. Firing a glider at a block (shown in 1) can move it, creating a "sliding block" that can act as a counter. The Game of Life has been used to simulate many algorithms and computers. It can simulate any Turing Machine and is theoretically as powerful as any computer of infinite memory, making it Turing Complete. The Turing completeness of the Game of Life allows Turing complete patterns to be made within it. These patterns are considered to be "non-elementary" as in can be deconstructed into a system of smaller interacting patters like those shown in sections 4.1, 4.2, 4.3, and 4.4. While these patterns may self-replicate, many of them take millions of generations to do so. This is both thanks to the tremendous size of these patterns and the speed at which information can travel. Any cell has only 8 neighbors, all of which are around its perimeter, so information (which is just the state of a certain pattern of cells) can travel at 1 cell per tick. This inefficiency has been embraced as the cellular automaton speed of light, denoted with c to parody the notation of the real speed of light.

The Game of Life is considered undecidable. If given an initial and future pattern, no algorithm exists that can determine whether the initial pattern will result in the future pattern. This undecidability also necessitates that some patterns remain chaotic forever. If this was not the case, an algorithm could emulate the Game of Life running the initial pattern until it stabilizes and will by then know whether the future pattern appeared. This parallels the halting problem, where no algorithm exists that can tell whether a program will run forever or terminate. Since the Game of Life is a Turing machine, it comes with the same limits as any other computer.

6 Algorithms

The Game of Life is not a difficult program to write and can be done with intermediate programming knowledge. The playfield is represented as a 2 dimensional array where each index is one of two constants, usually 1 and 0, to represent live and dead cells. Naturally, each index is a cell, and the neighbors are the indices where the row and column are offset by -1, 0, or 1. From there, the rules are just `if` statements. There are two methods for simulating a tick. The first method uses one array. If a cell is going to change states in the next generation, its index is appended to an array of cells to be born or an array of cells to die, depending on the next state of the cell. After all of the cells have been evaluated, the cells listed in the two arrays are updated. No computer can actually update every cell simultaneously, so they have to either save the cells into a new place in memory and update them after all the cells have been evaluated or to use a second 2D array, which is the second method. In the second method, once a cell is evaluated, its updated form is recorded in the second or "future" array, leaving the original array unaltered. Once all of the cells have been evaluated, the second array represents the next generation. The second array now represents the present and the first array is wiped and becomes the future array. The two arrays swap roles. There are many ways to reduce unnecessary computation. If a cell and all of its neighbors did not change on the last tick, then that cell is guaranteed to not change on the next tick. This data can be saved and the algorithm does not have to evaluate that cell. In practice, entire zones of the playfield are not evaluated. We can reduce computation further by only evaluating live cells and their neighbors, as dead cells have no hope of coming to life if they are not a neighbor of a live cell. The playfield has proved to be an issue for programmers. It cannot be infinite as a computer would need infinite memory to store that data. One could try to declare the cells outside the array as dead, but this can lead to inaccuracies when patterns cross the border. A common method is to think of the playfield as a torus instead of a finite plane. This way, patterns that travel off the edge of the playfield loop around to the other side. In code, this is represented as a toroidal array. Another option is to abandon the concept of a playfield and think of just live cells floating in a vacuum, with new live cells being born out of nothing. Live cells are represented as ordered pairs and their neighbors by variations of those ordered pairs. This removes the need to store data about dead cells (or "blanks") but can be computationally heavy as counting live neighbors becomes a hash table lookup. In theory, the only memory limit with this method is the number of live cells instead of total playfield size.

7 Code

We will code the Game of Life in a standard 2D array of live or dead cells. We will use Python and the pygame library to simulate the Game of Life interactively. The playfield can have any set of dimensions but we will use a square array of 50 by 50. The playfield begins entirely dead and the initial pattern is made by making specific cells alive by clicking on them. Pressing space shifts the program into a different mode where the configuration evolves according to the rules. Pressing space again resets the pattern. Below is the complete Python code.

```
# RULES FOR CONWAY'S GAME OF LIFE

# 1: the grid begins with a certain arrangement of live cells
# 2: the 8 cells that form the immediate perimeter of a cell are its neighbors
# (if the cell is on a corner or edge of the playfield, the cells on the other
# side of the playfield become its neighbors, so the playfield is a torus)
# 3: if a live cell has less than 2 neighbors, it dies next generation
# 4: if a live cell has more than 3 neighbors, it dies next generation
# 5: if a dead cell has exactly 3 neighbors, it is born next generation
# NOTE: deaths and births happen simultaneously
# NOTE: live cells are marked with 1, dead cells are marked with 0

# USER INSTRUCTIONS

# press L to toggle grid lines
# press esc to close the simulation

# in editor mode, click on cells to toggle them between live and dead
# when you are done with your configuration, press space to run the simulation
# in evolve mode, press space again to return to editor mode

# NEIGHBORS
# each function returns the neighbor of a specified cell in a certain direction

# find north neighbor
def find_N(grid,r,c):
    R = r-1
    if r == 0: R = h-1
    return grid[R][c]

# find south neighbor
def find_S(grid,r,c):
    R = r+1
    if r == h-1: R = 0
    return grid[R][c]

# find east neighbor
def find_E(grid,r,c):
    C = c+1
    if c == w-1: C = 0
    return grid[r][C]

# find west neighbor
def find_W(grid,r,c):
    C = c-1
    if c == 0: C = w-1
    return grid[r][C]

# find northeast neighbor
def find_NE(grid,r,c):
    R = r-1; C = c+1
    if r == 0: R = h-1
    if c == w-1: C = 0
    return grid[R][C]

# find northwest neighbor
def find_NW(grid,r,c):
    R = r-1; C = c-1
    if r == 0: R = h-1
    if c == 0: C = w-1
    return grid[R][C]

# find southeast neighbor
def find_SE(grid,r,c):
    R = r+1; C = c+1
    if r == h-1: R = 0
    if c == w-1: C = 0
    return grid[R][C]

# find southwest neighbor
def find_SW(grid,r,c):
    R = r+1; C = c-1
    if r == h-1: R = 0
    if c == 0: C = w-1
    return grid[R][C]

# OTHER FUNCTIONS
# return number of live neighbors of a cell
def live_neighbors(grid,pos):
    r,c = pos
    count = 0
    for neighbor in neighbors: count += neighbor(grid,r,c)
    return count

# return an empty wxh grid
def empty_grid(w,h):
    return [[0]*w for _ in range(h)]

# update grid to next generation
def next_gen(grid):
    newgrid = empty_grid(w,h) # next generation of grid starts empty (all dead)
    for r in range(h):     # go through every row and column
        for c in range(w):
            L = live_neighbors(grid,(r,c)) # L is the number of live neighbors
            if grid[r][c] == 0: # if the current cell is dead...
                if L == 3: newgrid[r][c] = 1 # come to life if condition met
            elif grid[r][c] == 1: # if the current cell is alive...
                if 2 <= L <= 3: newgrid[r][c] = 1 # stay alive if condition met
    return newgrid

# draw grid on screen
def draw_grid(grid,w,h,p):
    for r in range(h):
        for c in range(w):
            if evolve:
             pygame.draw.rect(screen, cols_evolve[grid[r][c]], (c*p,r*p,p,p))
            else:
             pygame.draw.rect(screen, cols_editor[grid[r][c]], (c*p,r*p,p,p))

# draw gridlines if permitted
def draw_lines(w,h,permit):
    if permit:
        for i in range(h):
            pygame.draw.line(screen, linecol, (0, i*p), (sw, i*p))
            for j in range(w):
                pygame.draw.line(screen, linecol, (j*p, 0), (j*p, sh))

# CONFIGURABLE VARIABLES
```

```
p = 35 # cell size in pixels
w = 50 # width of grid in cells
h = 50 # height of grid in cells
cols_editor = {0:(20,20,20), 1:(247, 208, 32)}
cols_evolve = {0:(20,20,20), 1:(208, 32, 247)}
linecol = (40,50,50)
gridlines = True # whether gridlines appear at the start or not

# SETUP

import pygame
sw = w*p # width of grid in pixels
sh = h*p # height of grid in pixels
neighbors = [find_N, find_S, find_E, find_W, find_NE, find_NW,
find_SE, find_SW] # list of all neighbor functions
grid = empty_grid(w,h) # initialize grid
mode = 'editor'
evolve = False
user_grid = grid
mousedown = False
generation = 0

pygame.init()
screen = pygame.display.set_mode((sw,sh))

# EVENT LOOP

while True:

    pygame.display.set_caption(f"Conway's Game of Life - {mode}")
    if evolve: mode = f'Generation {generation}'
    else: mode = 'Editor'

    if evolve: pygame.time.delay(40) # delays by speed
    for event in pygame.event.get():
        if event.type == pygame.QUIT: pygame.quit(); exit()

    keys = pygame.key.get_pressed()
    if keys[pygame.K_ESCAPE]: pygame.quit(); exit()
    if keys[pygame.K_l]: pygame.time.delay(30); gridlines = not gridlines

    draw_grid(grid,w,h,p)
    draw_lines(w,h,gridlines)
```

```
pygame.display.update()

if evolve:
    grid = next_gen(grid); generation += 1
    if keys[pygame.K_SPACE]:
        pygame.time.delay(30)
        evolve = False
        grid = empty_grid(w,h)
        generation = 0

else:
    if keys[pygame.K_SPACE]:
        pygame.time.delay(30)
        evolve = True
    if pygame.mouse.get_pressed()[0]:
        pygame.time.delay(60)
        pos = pygame.mouse.get_pos()
        pr = pos[0]; pc = pos[1]
        for r in range(h):
            for c in range(w):
                if p*c < pr < p*c+p and p*r < pc < p*r+p:
                    grid[r][c] ^= 1 # bitwise xor with 1
                    user_grid = grid; break
```