game of life

Guotai Shen

2023-11-03

Game of life

Conway's Game of Life (GOL) is a cellular automaton that mimics life forms on a two dimensional space, specifically grids of square "cells". Any cell has two states, dead or alive, represented by its brightness in our case. Although Game of life can be conducted on infinite R^2 space, for illustration purpose we show it on a finite chess board.

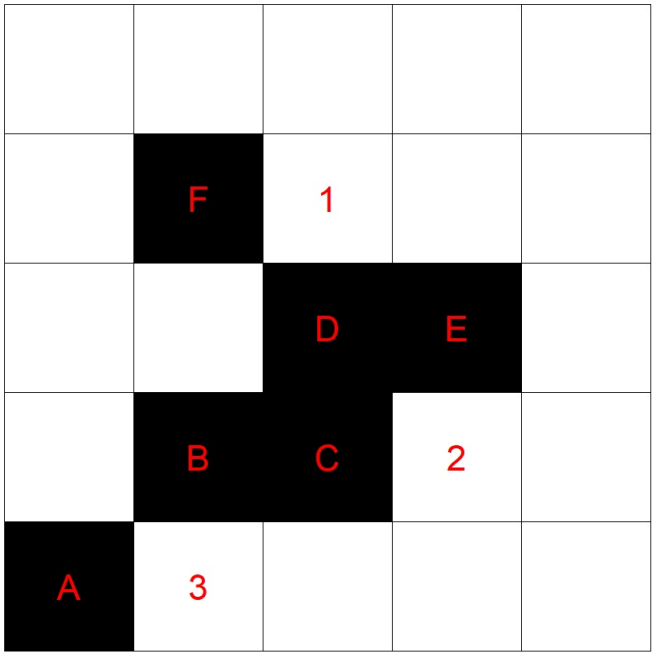
Rules of GOL

For any give cell, its next state depends on the current states of its surrounding 8 cells.

1. Any **live** cell with < 2 live neighbors **dies**, as if by **underpopulation**.
2. Any **live** cell with 2 or 3 live neighbors **lives on to the next generation**.
3. Any **live** cell with > 3 live neighbors **dies**, as if by **overpopulation**.
4. Any **dead** cell with exactly 3 live neighbors **becomes a live cell**, as if by **reproduction**.

Toy example

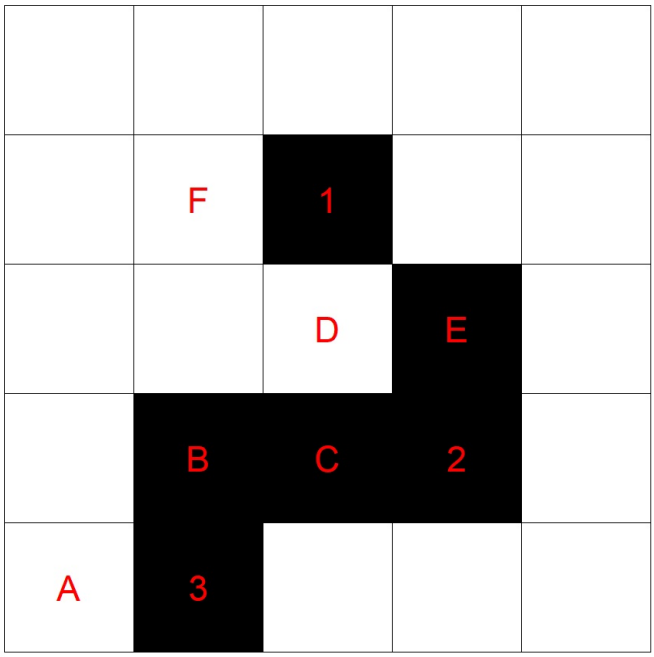
To better understand those rules, we can illustrate them using a toy model of a 5x5 grid space. 6 cells are "alive" with marks A through E, the other cells are currently dead in this state. A tile plot can illustrate the space better.



In the simple space illustrated above, among those "alive" cells, **A** and **F** have 1 "alive" neighbor, thus will **die** in next generation according to Rule 1 (underpopulation). **B**, **C** and **E** have 2 or 3 neighbors, so they will live through next generation (i.e. they will still be "alive" in next frame). Where **D** have more than 3 neighbors, it will died due to overpopulation.

Among those "dead" cells (white grids), grids marked with 1, 2 and 3 are surrounded by exactly 3 living cells, so according to Rule 4 they will be "alive" in next generation.

Hence we can conclude that the next generation of this toy model should look like the following:



Define functions

To implement the game of life, we need to first implement the most important function – that takes a grid space and evaluate the next generate, and output the grid for next generation. The function **`next_state()`** should do the following:

1. take the grid of the current generation and **pad one layer of dead cells** (so the detection range is always 8 cells).
2. initialized the result grid
3. for each cell in the grid count eight surrounding cells
4. based on the counting, determine the next state based on GOL
5. return the grid for next state

pad_zero()

This function takes a matrix and pad zeros to all sides of it.

```
pad_zero = function(grid){
  return(rbind(0,
               cbind(0, grid, 0),
               0
               ))
}
```

next_state()

We will implement the function as following:

```
next_state = function(grid){
  n = nrow(grid)

  ## check if input have the correct dimension
  if (n != ncol(grid)){
    stop("Input matrix must have equal length and width.")
  }

  ## pad 0's around the matrix
  grid.pad = pad_zero(grid)

  ## initialize next grid
  grid.next = matrix(0, nrow = n, ncol = n)

  ## for each cell count surrounding 8, and apply Rule
  for (i in 2:(n+1)){
    for (j in 2:(n+1)){
      neighbors = sum(grid.pad[(i-1):(i+1), (j-1):(j+1)]) - grid.pad[i,j]

      ## Apply the Rule
      if (grid.pad[i,j] == 1){
        grid.next[i-1,j-1] = ifelse(neighbors == 3 | neighbors == 2, 1, 0)
      } else {
        grid.next[i-1,j-1] = ifelse(neighbors == 3, 1, 0)
      }
    }
  }
  return(grid.next)
}
```

Examine the function

We can use our toy example to see if the function is performing well.

```
grid = matrix(c(0,0,0,0,0,
                0,1,0,0,0,
                0,0,1,1,0,
                0,1,1,0,0,
                1,0,0,0,0), nrow = 5, byrow = T)

grid
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    1    0
## [4,]    0    1    1    0    0
## [5,]    1    0    0    0    0
```

```
grid.next = next_state(grid)
grid.next
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0    0
## [2,]    0    0    1    0    0
## [3,]    0    0    0    1    0
## [4,]    0    1    1    1    0
## [5,]    0    1    0    0    0
```

The next-generation matrix is identical to the plot illustrated above.

Pre-process animation data

Now with the correct function to evaluate the GOL, we can pre-evaluate the grid to n steps later and use these data to create an animation of the GOL. This requires the `gganimate` package. But before plot the animation, we need to convert the list of grid matrices into one giant data_frame as input to ggplot.

```
## create initial grid
set.seed(123)
n = 100
percent = 0.2
num_generation = 60

vector.initial = sample(c(rep(1,round(percent*n^2)),
                          rep(0,n^2 - round(percent*n^2))))
grid.initial = matrix(vector.initial, nrow = n, byrow = T)

## create a list holding the grid at n-th generation
grid_states = list(grid.initial)

## evaluate grid states

for (i in 1:num_generation){
  grid_states[[i+1]] = next_state(grid_states[[i]])
}


## process the matrix into data for plotting (vectorize)
animation_frames = list()
for (i in 1:length(grid_states)){
  current_grid = grid_states[[i]]
  coordinates = expand.grid(x = 1:nrow(current_grid), y = 1:nrow(current_grid))
  current_state = as.vector(current_grid)

  generation_data = cbind(coordinates,
                          generation = i,
                          state = current_state)

  animation_frames[[i]] = generation_data
}
animation_data = do.call(rbind, animation_frames)
```

Animation

```
# Plot the animation using ggplot2 and gganimate
p = ggplot(animation_data, aes(x = x, y = y, fill = factor(state))) +
  geom_tile(color = "grey") +
  scale_fill_manual(values = c("0" = "white", "1" = "black")) +
  theme_minimal() +
  theme(panel.grid.major = element_blank(),
        panel.grid.minor = element_blank(),
        axis.text = element_blank(),
        axis.title = element_blank(),
        axis.ticks = element_blank(),
        legend.position = "none") +
  transition_states(generation, transition_length = 1, state_length = 1) +
  coord_fixed(ratio = 1) +
  labs(title = 'Generation: {closest_state}')

# Animate the plot
animate(p, nframes = num_generation, fps = 4, width = 400, height = 400)
```