Forest Fire Propagation: A Cellular Automata Approach

Assignment 1b | Probability & Statistics SA 2023-2024

Frova Davide

Abstract

The simulation that we are going to create will represent a forest made of different types of vegetation.

By spontaneous ignition or by a predifined starting point a forest fire born and spread, evolving in time in a Markov Chain manner.

Each cell will have a certain probability of ignition/burnout, which will be affected by it's previous state and the state of it's neighbours.

The probabilities will be affected by different factors:

- Specific cell type / state
- Near by cells types / states
- Natural events and characteristics
- Presents of puddles of water
- Presents of different kind of rivers
- Wind
- Altitude

Premise about code development

The development of some parts of the code that will follow were done with the help of the tool Github Copilot Chat, this was to fill in the missing syntax/functions knowledge about the R language. The tool was generating snippets of code to show examples on the usage of certain R functions, and it was mainly generating explanations about R functionalities, and as an help in finding not so easy to spot bugs and fixes to various problems.

Design and Implementation of the Forest Fire Model

Preparation-Step: Including the necessary external libraries

```
# Install and load the animation package
if (!require(animation)) install.packages("animation")
```

```
## Caricamento del pacchetto richiesto: animation
```

```
library(animation)
```

Defining the different types of cells

Here is the definition of the different cell types / states.

We will refer to *cell type* to indicate the type of the cell when we are looking at the forest model in a fixed time, and to *cell status/state* when we are in a on-going time frame, were the *type* of the cells is varying through time.

Pseudocode:

```
cell_states is a dictionary/set of states
  1. water
  2. dryGrass
  3. denseTrees
  4. burning
  5. normalForest
  6. burned
```

R code:

```
cell_states <- c(
  "Water", # 1
  "DryGrass", # 2
  "DenseTrees", # 3
  "Burning", # 4
  "NormalForest", # 5
  "Burned" # 6
)
```

We preferred using the indexes to refer to a cell_state in the following code sections. The cell_states dictionary names will be used in the plot visualization, as a legend for the color coding.

Initialization of the forest grid

The forest will be represented by an n*m grid / matrix, each cell has it's type defined (index in the dictionary of types) and it's own probability of ignition/burnout.

In this function we will create the matrix and set the default value of the status and probability.

Pseudocode

```
initialize_grid is a function with parameters (n rows, m cols)
  grid is a matrix n by m
  foreach element of grid
    set the status field to normalForest
    set the probability to 0.0
  return the grid
```

R code:

```
initialize_grid <- function(rows, cols) {
  # Create a list of matrices
  # status will store the status of the cells
  # prob   will store the probability of the cells
  grid <- list(
    "status" = matrix(nrow = rows, ncol = cols),
    "prob" = matrix(nrow = rows, ncol = cols)
  )

  # Foreach element set the status and probability
  for (y in 1:rows) {
    for (x in 1:cols) {
```

```
      grid$status[y, x] <- 5 # Normal Forest
      grid$prob[y, x] <- 0.0
    }
  }
  return(grid)
}
```

In the R code we have different ways of creating a matrix that for each cell contains 2 fields.

In this case we chose to create a matrix for the statuses and a matrix for the probabilities. Then we store these two matrices in a list called *grid*

Test of the function

```
initial_forest <- initialize_grid(3, 3)
initial_forest  # Original Values
```

```
## $status
##      [,1] [,2] [,3]
## [1,]    5    5    5
## [2,]    5    5    5
## [3,]    5    5    5
##
## $prob
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
## [3,]    0    0    0
```

```
# Set the status matrix with the names of the states
initial_forest$status <- matrix(
  sapply(
    initial_forest$status,
    function (val) cell_states[val]),
    nrow = nrow(initial_forest$status),
    ncol = ncol(initial_forest$status
  )
)

initial_forest  # Names instead of values
```

```
## $status
##      [,1]           [,2]           [,3]
## [1,] "NormalForest" "NormalForest" "NormalForest"
## [2,] "NormalForest" "NormalForest" "NormalForest"
## [3,] "NormalForest" "NormalForest" "NormalForest"
##
## $prob
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
## [3,]    0    0    0
```

Filling the grid randomly

Now we generated a grid with default values inside. We could run a simulation based on it, but in our case we want more interesting characteristics to observe. That's why we will develop some functions to fill the forest grid in random and semi-random ways with other values for the cell types.

Random values for the cell states

A basic first approach is to assign to each cell of the grid a random value from the cell_states dictionary

Pseudocode

```
random_fill is a function with parameters (forest_grid)
  foreach cell in forest_grid
    new_status is a random pick from cell_states
    set the cell "status" field to new_status
```

R code:

```
random_fill <- function(forest_grid){
  max_y <- dim(forest_grid$status)[1]
  max_x <- dim(forest_grid$status)[2]
  for (y in 1:max_y) {
    for (x in 1:max_x) {
      # Sample from the indexes of the cell_states
      status <- sample(1:length(cell_states), 1)
      forest_grid$status[y, x] <- status
    }
  }
  return (forest_grid)
}
```

Test of the function

```
initial_forest <- initialize_grid(3, 3)
filled_forest <- random_fill(initial_forest)
filled_forest
```

```
## $status
##      [,1] [,2] [,3]
## [1,]    5    5    1
## [2,]    1    5    3
## [3,]    6    2    1
##
## $prob
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
## [3,]    0    0    0
```

We will further develop more advanced ways of filling the grid, but for now this is enough to continue with the definition of other core functionalities.

Design of the plot_grid function

Now that we have a way of randomly generate the forest grid, we can think of a way to visualize it.

Printing it with the matrix R representation is not the best as we saw before, so we are going to develop a function to plot our matrix in a color coded way.

Each color will represent a different type of cell.

Color palette

First we define a color palette to represent the different cells

Pseudocode

```
cell_states_colors is a dictionary containing colors
  1. Water color
  2. Dry grass color
  3. Dense trees color
  4. Buring/Fire color
  5. Normal forest color
  6. Burned color
```

R code

```r
cell_states_colors <- c(
  "#0000FF", # Blue / Water
  "#ADFF2F", # GreenYellow / DryGrass
  "#006400", # DarkGreen / DenseTrees
  "#FF0000", # Red / Burning
  "#008000", # Green / NormalForest
  "#201F1F"  # DarkGray / Burned
)
```

We represent the colors using the hexadecimal values, one for each of the cell types.

Plot grid function

Now we can define a function that will plot a representation of our matrix, that will create a rectangle/square for each cell, with the corresponding status color.

Pseudocode

```
plot_grid is a function with parameters (forest_grid, cell_states_colors)
  foreach row of forest_grid
    row_of_cells is an empty list
    foreach cell in the current row
      add a rectangle with the corresponding cell state color to the list
    plot the row_of_cells with the rectangles in line
    go to a new line in the plot
```

The R code implementation varied a lot during the development. The main changes where:

- Adding a title with various information to the plot
- Optimization in terms of speed of the plotting function
- Optimization of the quality of the produced plot

This was the first implementation

```r
plot_grid <- function(forest_grid) {
  # I flip&transpose the grid for better visualization
  # and consistency with the print of the values (0,0 top left)
  flipped_grid <- apply(forest_grid, 2, rev)
  transposed_again <- t(flipped_grid)

  # Get unique values to only take the colors needed
  # (image() will be angry otherwise)
```

```
  colors <- sort(unique(as.vector(forest_grid)))

  # Map each cell status to its color
  colors <- sapply(colors, function(val) cell_states_colors[val])

  # Print the image of the grid
  image(transposed_again, col = colors)
}
```

This is the final implementation with the above listed changes

R code

```
plot_grid <- function(forest_grid, plot_title = NULL) {
  # We will be working only on the status matrix
  forest_grid <- forest_grid$status

  # Increase the size of the grid matrix
  # This is to achieve a plot/graphic with higher quality
  # Each cell is now 10 cells
  # thanks to Github Copilot Chat
  enlarged_forest_grid <-
    kronecker(
      forest_grid,
      matrix(1, nrow = 10, ncol = 10)
    )

  # Get the number of rows and columns of the enlarged matrix
  nrow_enlarged <- dim(enlarged_forest_grid)[1]
  ncol_enlarged <- dim(enlarged_forest_grid)[2]

  # Create an empty plot
  plot(
    0, 0,
    type = "n",
    xlim = c(0, ncol_enlarged),
    ylim = c(0, nrow_enlarged),
    xlab = "", ylab = "", xaxt = "n", yaxt = "n",
    asp = 1
  )

  # Add the title to the plot
  if (!is.null(plot_title)) {
    title(main = plot_title)
  }

  # Create a color matrix
  # This will map each cell status to its cell_status_color
  color_matrix <- matrix(
    cell_states_colors[enlarged_forest_grid],
    nrow = nrow_enlarged,
    ncol = ncol_enlarged,
    byrow = TRUE
  )
```

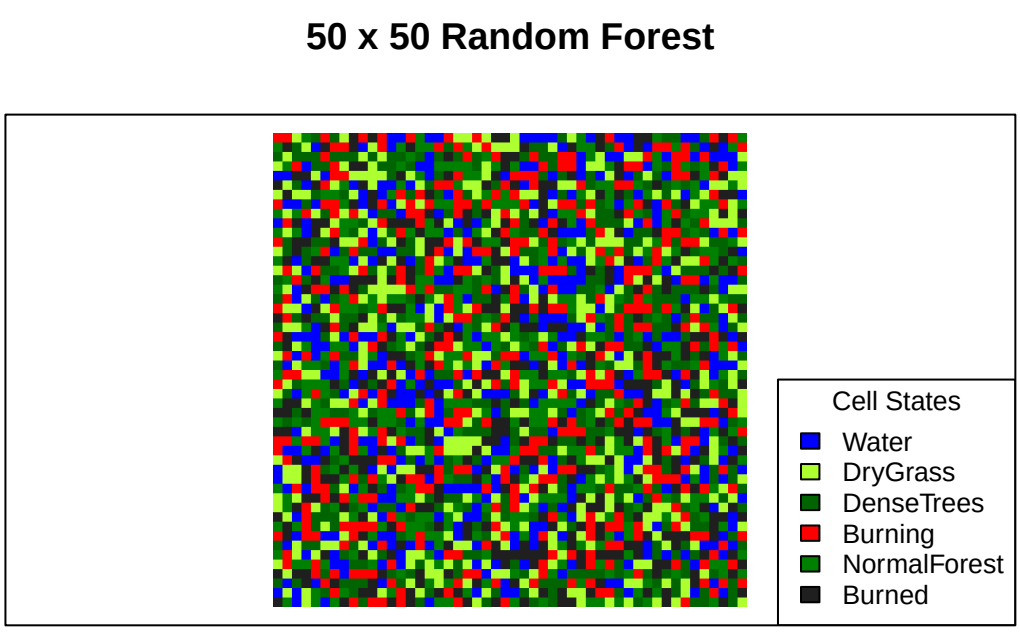
```
  # Convert the color matrix to an image
  # Using raster for faster plotting
  img <- as.raster(color_matrix)

  # Draw the image
  rasterImage(img, 0, 0, ncol_enlarged, nrow_enlarged)

  # Add a legend
  legend(
    "bottomright",  # Position of the legend
    legend = cell_states,  # Labels for the legend items
    fill = cell_states_colors,  # Colors for the legend items
    cex = 0.8,  # Size of the legend items
    title = "Cell States"  # Title of the legend
  )
}
```

Test of the function

```
initial_forest <- initialize_grid(50, 50)
filled_forest <- random_fill(initial_forest)
plot_grid(filled_forest, "50 x 50 Random Forest")
```



C. Develop the neighbours function

Create pseudocode to identify neighboring cells of a given point.

```r
# Example output:
# [[1]]
#    state coords.x coords.y
#        3        2        1

# [[2]]
#    state coords.x coords.y
#        4        1        2

# [[3]]
#    state coords.x coords.y
#        1        2        2

# forest_grid object
neighbours <- function(forest_grid, cell) {
  x <- cell[2]
  y <- cell[1]

  max_x <- dim(forest_grid$status)[2]
  max_y <- dim(forest_grid$status)[1]

  result <- list(
    c(y - 1, x - 1), # Top left
    c(y, x - 1), # Top center
    c(y + 1, x - 1), # Top right
    c(y - 1, x), # Center left
    c(y + 1, x), # Center right
    c(y - 1, x + 1), # Bottom left
    c(y, x + 1), # Bottom center
    c(y + 1, x + 1) # Bottom right
  )

  valid_result <- list()

  for (negh in result) {
    if (negh[1] > 0 && negh[2] > 0 && negh[2] <= max_x && negh[1] <= max_y) {
      negh_obj <- c(
        "state" = forest_grid$status[negh[1], negh[2]],
        "coords" = c(
          "x" = negh[2],
          "y" = negh[1]
        )
      )

      valid_result <- append(valid_result, list(negh_obj))
    }
  }

  return(valid_result)
}
```

D. Construct the propagate function

Draft pseudocode to manage fire propagation based on: - The type of area - Its current state - Neighboring cell states.

```
# Example output:
# [1] 0.3 0.0 0.0



# "Water",          # 1
# "DryGrass",       # 2
# "DenseTrees",     # 3
# "Burning",        # 4
# "NormalForest"    # 5

# Cell:
# cell <- c(
#   "state" = 1--5,
#   "coords" = c(
#     "x" = 1,
#     "y" = 1
#   )
# )
propagate <- function(cell, neighbours) {
  if (cell[["state"]] != 4) {
    return(sapply(neighbours, function(val) 0.0))
  }

  neghs_probs <- sapply(neighbours, function(neg) {
    return(
      switch(neg[["state"]],
        "1" = 0.0, # max = 0.0
        "2" = 0.125, # max = 1.0
        "3" = 0.05, # max = 0.4
        "4" = 0.0, # max = 0.0
        "5" = 0.0875, # max = 0.7
        "6" = 0.0 # max = 0.0
      )
    )
  })

  return(neghs_probs)
}
```

E. Design the update_grid function:

Develop pseudocode to manage fire dynamics and grid updates using the neighbours and propagate functions.

```
# "Water",          # 1
# "DryGrass",       # 2
# "DenseTrees",     # 3
# "Burning",        # 4
# "NormalForest"    # 5
# "Burned"          # 6
```

```r
update_grid <- function(forest_grid) {
  for (y in 1:nrow(forest_grid$status)) {
    for (x in 1:ncol(forest_grid$status)) {
      # Get the current cell state
      cell_state <- forest_grid$status[y, x]
      cell_prob <- forest_grid$prob[y, x]

      # If the cell could burn, generate a random number and check with its probability
      # if so, set them to burning and reset their prob
      if (cell_state == 2 || cell_state == 3 || cell_state == 5) {
        rand_value <- runif(1)
        if (rand_value <= cell_prob) {
          forest_grid$status[y, x] <- 4
          forest_grid$prob[y, x] <- 0.0

          cell_state <- 4
          cell_prob <- 0.0
        }
      } else if (cell_state == 4) {
        # If the cell is burning, and the random value gets the prob
        # set it to burned and reset its prob, otherwise increase the prob of going out

        rand_value <- runif(1)
        if (rand_value <= cell_prob) {
          forest_grid$status[y, x] <- 6
          forest_grid$prob[y, x] <- 0.0

          cell_state <- 6
          cell_prob <- 0.0
        } else {
          # Increase the probability of going out
          forest_grid$prob[y, x] <- forest_grid$prob[y, x] + 0.3
          cell_prob <- cell_prob + 0.1
        }
      }

      # Get the neighbours of the current cell
      neghs <- neighbours(forest_grid = forest_grid["status"], cell = c(y, x))

      # Create an "object" with the data needed from the propagate function
      propagate_cell <- c(
        "state" = cell_state,
        "coords" = c(
          "x" = x,
          "y" = y
        )
      )

      # Compute the probabilities of the propagate of the neighbours cells
      probs_propagate <- propagate(propagate_cell, neghs)

      # Foraech of the neighbours for which we have the probs. of ignition
      # we sum it to the current prob. values of the forest_grid
```

```
      # (Markov Chain) t_n = t_n-1 + added_prob
      for (negIndex in 1:length(neghs)) {
        forest_grid$prob[neghs[[negIndex]][["coords.y"]], neghs[[negIndex]][["coords.x"]]] <-
          forest_grid$prob[neghs[[negIndex]][["coords.y"]], neghs[[negIndex]][["coords.x"]]] + probs_pr
      }
    }
  }

  return(forest_grid)
}
```

```
plot_probs_and_states <- function(forest_grid, probs = TRUE) {
  flipped_grid <- apply(forest_grid$status, 2, rev)

  flipped_probs <- apply(forest_grid$prob, 2, rev)

  # Create a blank plot
  plot(1, 1,
    xlim = c(1, ncol(flipped_grid)), ylim = c(1, nrow(flipped_grid)),
    type = "n", xlab = "", ylab = "", xaxt = "n", yaxt = "n"
  )

  # Draw rectangles for each cell
  for (i in 1:nrow(flipped_grid)) {
    for (j in 1:ncol(flipped_grid)) {
      rect(j - 0.5, i - 0.5, j + 0.5, i + 0.5, col = cell_states_colors[flipped_grid[i, j]])
    }
  }

  # Add probabilities on top of the rectangles
  for (i in 1:nrow(flipped_probs)) {
    for (j in 1:ncol(flipped_probs)) {
      text(j, i, labels = ifelse(probs, flipped_probs[i, j], flipped_grid[i, j]), cex = 1.5)
    }
  }
}
```

Adanced grid filling functions

```
# Parametric function that enables the user to fill the forest with a specific function "shape"
# and a specific probability function, with a desired cell_type
fill_grid <- function(forest_grid, pos_func, prob_func, cell_type, thickness = 0.5, flip_orientation = 
  max_y <- dim(forest_grid$status)[1]
  max_x <- dim(forest_grid$status)[2]

  for (y in 1:max_y) {
    for (x in 1:max_x) {
      if (pos_func(y, x, max_y, thickness, flip_orientation)) {
        forest_grid$status[y, x] <- ifelse(
          cell_type == -1,
          forest_grid$status[y, x],
          cell_type
        )
```

```
        forest_grid$prob[y, x] <- prob_func(x, y)
      }
    }
  }

  return(forest_grid)
}

# Function ultra-parametric that enables the user to fill the forest with a specific function "shape"
# y, x are the coordinates of the point to be tested,
# min_y, max_y, max_x will be used to handle the ratio/span of the function in the grid
# func is the function used that will produce the value to compare with the thickness
# thickness is the sensitivity/distance from the y,x coordinate and the actual function value
# Returns true or false if the coordinate is or not generated by the function
func_fill <- function(y, x, min_y, max_y, max_x, func, thickness) {
  # Scale y to the range of the sine function.
  # Put the 0 of the sin in the middle of my matrix
  y_scaled <- ((y - min_y) / (max_y - min_y)) * 2 - 1

  # Scale x to the range of the sine function.
  x_scaled <- (x - max_x / 2) / (max_x / 2)

  # Calculate the y value of the sine wave at position x
  y_sin <- func(x_scaled)

  # Check if the scaled y value is close to the y value of the sine wave
  return(abs(y_scaled - y_sin) < thickness)
}

rand_pos <- function(y, x, max_y, thickness, flip_orientation = FALSE) {
  if (flip_orientation) {
    # Randomly change the function orientation
    temp <- y
    y <- x
    x <- temp
  }
  return(
    func_fill(
      y = y,
      x = x,
      min_y = 0,
      max_y = max_y,
      max_x = pi,
      func = function(val) runif(1, -max_y, max_y),
      thickness = thickness # 0.2
    )
  )
}

rand_max_x <- pi * sample(10:50, 1)
rand_min_y <- sample(2:20, 1)

exp_pos <- function(y, x, max_y, thickness, flip_orientation = FALSE) {
```

```
  if (flip_orientation) {
    # Randomly change the function orientation
    temp <- y
    y <- x
    x <- temp
  }
  return(
    func_fill(
      y = y,
      x = x,
      min_y = max_y * rand_min_y,
      max_y = max_y,
      max_x = rand_max_x,
      func = exp,
      thickness = thickness # 0.3
    )
  )
}

# Wrapper function for the func_fill
# It will produce a sin_wave function (wavy river like)
sin_pos <- function(y, x, max_y, thickness, flip_orientation = FALSE) {
  if (flip_orientation) {
    # Randomly change the function orientation
    temp <- y
    y <- x
    x <- temp
  }
  return(
    func_fill(
      y = y,
      x = x,
      min_y = 0,
      max_y = max_y,
      max_x = rand_max_x,
      func = sin,
      thickness = thickness # 0.3
    )
  )
}
```

F. Draft the simulate function

Provide pseudocode to orchestrate the entire simu- lation over multiple iterations using the previously developed functions.

```
rand_max_x <- pi * sample(10:50, 1)
rand_min_y <- sample(2:20, 1)

simulate <- function(t_max, N, grid_rows, grid_cols) {
  # N simulations, until t_max iterations

  for (i in 1:N) {
    paste("Simulation", i)
```

```r
# Initialize the grid full of normal forest
forest_grid <- initialize_grid(grid_rows, grid_cols)

# CUSTOMIZABLE PART -------------------------

# Add random dry grass with some small self ignition probability
forest_grid <- fill_grid(
  forest_grid = forest_grid,
  pos_func = rand_pos,
  prob_func = function(y, x) 0.0002, # Starting Probability (self-ignition prob)
  cell_type = 2, # Dry grass
  thickness = runif(1, 0, 20) # Higher value = Higher population
)

# Add random dense trees with some small self ignition probability
forest_grid <- fill_grid(
  forest_grid = forest_grid,
  pos_func = rand_pos,
  prob_func = function(y, x) 0.00002, # Starting Probability (self-ignition prob)
  cell_type = 3, # Dense trees
  thickness = runif(1, 0, 20) # Higher value = Higher population
)

#  Randomize the values of the exp_pos function
rand_max_x <- pi * sample(10:50, 1)
rand_min_y <- sample(2:20, 1)

# Adding a river with a random function plotted (exp or sin)
# I kept them separated and not param only the func_pos to be able to modify them separately
rand_value_river <- runif(1)
water_type <- 0
if (rand_value_river < 0.33) {
  # Exp function
  forest_grid <- fill_grid(forest_grid, exp_pos, function(y, x) 0.0, 1, runif(1, 0.05, 0.3), ifelse
  water_type <- 0
} else if (rand_value_river >= 0.33 && rand_value_river < 0.66) {
  # Random sin river
  forest_grid <- fill_grid(forest_grid, sin_pos, function(y, x) 0.0, 1, runif(1, 0.05, 0.3), ifelse
  water_type <- 1
} else {
  # Random rain puddles
  forest_grid <- fill_grid(forest_grid, rand_pos, function(y, x) 0.0, 1, runif(1, 0.05, 10), ifelse
  water_type <- 2
}

# Random starting point burning
forest_grid$status[sample(1:grid_rows, 1), sample(1:grid_cols, 1)] <- 4

# Define the animation function

for (t in 1:t_max) {
  # Print the current iteration
  # paste("Iteration", t)
```

```
      # Update the grid
      forest_grid <- update_grid(forest_grid)

      # Plot the grid
      plot_grid(
        forest_grid,
        paste(
          "Simulation n. ", i, " / ", N, "\nStep n. ", t, " / ", t_max,
          "\n Water type: ", ifelse(water_type == 0, "Exp River", ifelse(water_type == 1, "Sin River",
        )
      )
      print(paste("T: ", t, " / ", t_max))
    }
    print(paste(i, "| Simulation Completed"))

    # Save the animation as a GIF
    # saveGIF(ani.fun(), movie.name = paste("forest_simulation.gif"), interval = 0.2)
  }
}

# simulate(
#   t_max = 20,
#   N = 2,
#   grid_rows = 100,
#   grid_cols = 100
# )

# Gif
# saveGIF(
#   simulate(
#     t_max = 50,
#     N = 5,
#     grid_rows = 100,
#     grid_cols = 100
#   ),
#   movie.name = paste("forest_simulation.gif"),
#   interval = 0.2
# )

# Movie
# saveVideo(
#   simulate(
#     t_max = 10,
#     N = 5,
#     grid_rows = 50,
#     grid_cols = 50
#   ),
#   video.name = paste("forest_simulation.mp4"),
#   interval = 0.2
# )
```