Machine Learning in the Face of Computational Complexity:

Reversing Conway's Game of Life

David Augustine

Contents

Foreword

This report has been created as part of an R project requirement for the completion of HarvardX's Data Science Professional Certificate program. HarvardX is a University-wide strategic initiative at Harvard University, designed to facilitate online education by faculty of Harvard for students everywhere.

Chapter 1

Introduction

Of the many reasons to which machine learning's increasing importance has been attributed, one more likely to be overlooked than the others is the fact that there exist *computationally complex* problems that, although solvable and understood, cannot be addressed in any practical or meaningful way using today's technology.

In this report, we present, analyze and model a well-known example of one such problem, known as *Conway's Game of Life (GoL)*. This game is a specific type of model belonging to the more general class of models referred to as *cellular automata*, and is characterized and distinguished from the other models in this class by the particular set of rules that define it.

Any instance of Conway's GoL can be defined as a sequence of square grids of at least 9 cells, such that each cell is either living (has a value of 1) or dead (has a value of 0), and with each grid in the sequence being completely determined by the grid preceding it using the rule that if a cell is living and has 3 or 4 living cells in its neighborhood (defined as the square consisting of the 9 cells surrounding and including the cell), or is dead and has exactly 3 living cells in its neighborhood, then the corresponding cell in the same position in the subsequent grid in the sequence will be living; otherwise, that corresponding cell will be dead.

Despite this simple rule, Conway's GoL cannot be reversed easily or quickly for sufficiently large grid sizes. Reversing the GoL by determining the grids preceding a given grid is therefore the problem we concern ourselves with in this report.

Some properties of GoL grids that we should point out before proceeding are that there exist families of grids for which no preceding grids exist (these are referred to as *Gardens of Eden* and can be characterized by the unique constituent *orphans* - which can themselves be thought of as smaller "irreducible" Gardens of Eden - that define them), that there are certain sequences of grids that actually repeat themselves in an infinite loop every so often (the exact number of time steps between repetitions being their *periods*), that some grids can have multiple grids from which they evolve (meaning the rules defining The Game of Life fail to satisfy *injectivity* when viewed as a function), and that we assume the cells of the grids wrap around, so that the first row follows after the last row and the first column follows after the last column, for example.

It's worth mentioning that, as a class of objects widely studied in theoretical computer science, the importance of being able to quickly and efficiently solve problems involving cellular automata can be suggested by noting that many scholars have raised the question of whether or not our own universe is itself a cellular automaton.

Our work will be done entirely in R. We'll first attempt to solve a couple simple examples of The Game of Life using brute force, before deducing it'd be impractical to solve a collection of 50,000 25x25 examples of The GoL (our testing dataset) using this approach due to runtime constraints, and instead resorting to two of the most basic machine learning methods known to accomplish this. To construct our machine learning models, we'll be training them on a separate dataset of 50,000 25x25 GoL configurations (our training dataset). For all examples in both datasets, the number of steps between starting and ending grids can be anywhere from 1 to 5.

Chapter 2

Methods and Analysis

2.1 Initialization

Before working with any data, we must first initialize our work environment within R by loading several different packages and defining multiple objects to be used throughout the subsequent steps. We do this by executing the following code (note that not all packages loaded are used for this project; those listed are simply the ones I generally load before doing work in R).

```
# Define names of packages to load for scripts. Not all are actually necessary.

Packages <- c("bigmemory", "bigstatsr", "broom", "caret", "compiler", "caTools",
  "data.table", "doParallel", "doSNOW", "dplyr", "dslabs", "e1071",
  "fastAdaboost", "foreach", "formatR", "future", "gam", "genefilter", "ggplot2",
  "ggrepel", "gridExtra", "HistData", "ipc", "kernlab", "knitr", "Lahman",
  "lpSolve", "lubridate", "MASS", "matrixStats", "mvtnorm", "naivebayes",
  "parallel", "pdftools", "promises", "purrr", "randomForest", "ranger",
  "Rborist", "RColorBrewer", "recommenderlab", "recosystem", "reshape2", "rlist",
  "ROSE", "rpart", "rpart.plot", "rtweet", "rvest", "scales", "snow", "stringr",
  "svMisc", "svSocket", "textdata", "tibble", "tidyr", "tidytext", "tidyverse",
  "tree", "zoo")

# Download and install any packages not already installed and then load them.

for(p in Packages){
  if(!require(p, character.only = TRUE)){install.packages(p,
    character.only = TRUE, repos = "http://cran.us.r-project.org")}
  library(p, character.only = TRUE)
}

# Set the number of logical processors to be used for parallel computing as well
```

```r
# as the corresponding cluster of child processes, and register the back-end for
# the cluster.

logical_CPUs <- detectCores(logical = TRUE)

reverse_GoL_cluster <- makeCluster(logical_CPUs)

registerDoParallel(reverse_GoL_cluster)

# Define a function that takes a game grid in vector form and creates a matrix
# consisting of 0s and 1s, the locations of which being based on both the
# relationship between the indices of cells when game grids are represented as
# vectors and the indices of cells when game grids are represented as matrices as
# well as the relationship between the indices of cells when game grids are
# represented as matrices and the indices of their neighboring cells when game
# grids are represented as matrices.

create_aux_matrix <- function(S.p1){
  S.length <- length(S.p1); S.mtrx.dim <- sqrt(S.length)
  if(S.mtrx.dim %% 1 != 0){
    print("Provided vector can't be expressed as a square matrix!")
  }else{
    M_aux <- matrix(nrow = S.length, ncol = S.length)
    for(k in 1:S.length){
      # Define index pair (i,j) of grid in matrix form corresponding to index k
      # of grid in vector form.
      j <- ifelse(k %% S.mtrx.dim == 0, S.mtrx.dim, k %% S.mtrx.dim)
      i <- ((k - j) + S.mtrx.dim)/S.mtrx.dim
      # Define indices for 9 cells in neighborhood of each cell, including the
      # cell itself, in grid in matrix form.
      for(J in ifelse((j - 1):(j + 1) %% S.mtrx.dim == 0, S.mtrx.dim,
        (j - 1):(j + 1) %% S.mtrx.dim)){
        for(I in ifelse((i - 1):(i + 1) %% S.mtrx.dim == 0, S.mtrx.dim,
          (i - 1):(i + 1) %% S.mtrx.dim)){
          # Define indices K of grid in vector form corresponding to index pairs
          # (I,J) of grid in matrix form.
          K <- (I - 1)*S.mtrx.dim + J
          M_aux[k, K] <- 1
        }
      }
    }
    # Define a value of 0 to elements not assigned a value of 1.
    M_aux[is.na(M_aux)] <- 0
    return(M_aux)
  }
}

# Define a function that takes a game grid in vector form, evolves the grid
# forward one time step, and returns the resulting grid in vector form.

evolve_S <- function(S.p2){
  N <- create_aux_matrix(S.p2) %*% S.p2
  ifelse(S.p2 == 1, ifelse(N %in% c(3, 4), 1, 0), ifelse(N == 3, 1, 0))
}

# Define a function that takes a game grid in vector form, evolves the game grid
# forward the specified number of time steps, and returns the resulting grid in
# vector form.

generate_S_stop <- function(S_start.p, delta.p1){
  S_temp <- S_start.p
  for(j in 1:delta.p1){
    S_temp <- evolve_S(S_temp)
  }
  return(S_temp)
}

# Define a function that takes a game grid in vector form, that game grid's
# corresponding neighborhood sum matrix in vector form and a second game grid in
# vector form and checks the relationships between the three objects, returning
# TRUE if there are no inconsistencies between any of the objects' elements
# (assuming the objects follow the rules of Conway's Game of Life).

check_S_parent <- function(S_parent.p, S_child.p, N.p){
  all((S_child.p == 0 | (N.p %in% c(3, 4) & (N.p != 4 | S_parent.p == 1))) &
    (S_child.p == 1 | (N.p != 3 & (N.p != 4 | S_parent.p == 0))) &
    ((N.p %in% 1:9 | S_parent.p == 0) & (N.p %in% 0:8 | S_parent.p == 1) &
    S_parent.p %in% c(0, 1)))
}

# Define a function that takes a game grid in vector form, recursively solves it
# backwards the specified number of times, and then returns the first resulting
# grid ancestor that it generates as a solution in vector form.

solve_S_ancestor <- function(i.p2, S_descendent.p, t.p){
  # Set seed for the sake of reproducibility.
  set.seed(i.p2, sample.kind = "Rounding")
  # Define permutation of indices of ancestor grid's cells that are non-trivial
  # and determine the living cells in S_stop.
  indices <- M[which(S_descendent.p == 1),] %>%
    apply(1, function(x){which(x == 1)}) %>% as.vector() %>% unique() %>% sample()
  # Define the set of candidate values for each cell in parent grid, based on
  # whether corresponding cells in S_stop are living or dead.
  S.range <- lapply(as.list(1:length(S_descendent.p)), function(s){
    if(s %in% indices){
      c(0, 1)
    }else{
      0
    }
  })
  # Initialize index to 1 for all cells in parent grid, so that, to start, the
  # first element of each set in S.range is assigned to the value of its
  # corresponding cell in parent grid.
  S.index <- rep(1, times = length(S_descendent.p))
  # Rather than using for() loop, use repeat() loop, as number of iterations of
  # loop is unknown at execution.
  repeat{
    # If complete_ind[1] has been reassigned a value of 1 by the start of each
    # iteration of the repeat() loop, exit the loop, as this indicates a solution
    # has already been found by one of the other child processes in the cluster
    # and there is therefore no need for this particular process to continue
    # searching for a solution.
    if(complete_ind[1] == 1){return(NULL)}
    # Assign values to parent grid for current iteration.
    S_parent <- mapply(function(x, y){x[[y]]}, S.range, S.index)
    # Define vector of neighborhood sums corresponding to resulting parent grid.
    N <- M %*% S_parent
    if(check_S_parent(S_parent, S_descendent.p, N) == TRUE){
      # Record ancestor grid in log of reversal process, then either assign
      # indicator variable a value of "Y" so that function is exited or move on
      # to solving previous ancestor grid.
      game_log[[t.p]] <<- list(name = paste("S_", t.p - 1, sep = ""),
        value = S_parent)
      if(t.p == 1){
        solution_found <<- "Y"
      }else{
        solve_S_ancestor(i.p2, S_parent, t.p - 1)
      }
    }
    m <- match(TRUE, S.index[indices] < 2)
    # If either a solution is found or there are no more possible values to try
    # for parent grid, exit the current repeat() loop, otherwise define the
```

```
    # indices to be used to determine the next candidate parent grid and continue.
    if(solution_found == "Y" | is.na(m)){
      break
    }else{
      S.index[indices][index(S.index[indices]) < m] <- 1
      S.index[indices][m] <- S.index[indices][m] + 1
    }
  }
}

# Define a function that takes a game grid in vector form then first initializes
# a few objects before calling on solve_S_ancestor() to either return the first
# resulting grid ancestor sequence generated in vector form or return NULL if no
# such sequence exists.

solve_S_start <- function(i.p1, S_stop.p1, delta.p2){
  solution_found <<- "N"
  game_log <<- list()
  M <<- create_aux_matrix(S_stop.p1)
  solve_S_ancestor(i.p1, S_stop.p1, delta.p2)
  if(solution_found == "Y"){complete_ind[1] <- 1}
  return(game_log)
}

# Define a function that simultaneously executes the solve_S_start() function in
# parallel on each logical processor.

parallelize_solve_S_start  <- function(S_stop.p2, delta.p3){
  foreach(i = 1:logical_CPUs,
    .export = c("solve_S_start", "complete_ind", "create_aux_matrix",
    "solve_S_ancestor", "%>%", "check_S_parent", "index")) %dopar% {
      solve_S_start(i, S_stop.p2, delta.p3)
  }
}

# Define the final outside function that either returns the first solution found
# between the different child processes executed in parallel or returns a message
# indicating no solution exists.

aggregated_solve_S_start <- function(S_stop.p3, delta.p4){
  # Initialize the 1 x 1 matrix, complete_ind, the value of which will be shared
  # between all child processes executing solve_S_start() in parallel.
  complete_ind <<- FBM(1, 1, init = 0)
  # Clean up the list of outputs returned by solve_S_start() by removing all NULL
  # elements (including those that are nested).
  x <- parallelize_solve_S_start(S_stop.p3, delta.p4) %>%
    list.clean(., recursive = TRUE)
  if(complete_ind[1] == 1){
    # If complete_ind[1] == 1, then at least one of the remaining elements of x
    # will consist of a complete sequence of ancestor grids, and the below
    # statement will return the first such element of x for which this is true.
    return(x[[match(delta.p4, lapply(x, function(y){length(y)}) %>% cbind())]])
  }else{
    # If complete_ind[1] != 1, then none of the child processes executing
    # solve_S_start() in parallel found a solution, so print the below message.
    print("No solution exists!")
  }
}

# Unzip and import train and test datasets into R.

data.dir.path <- paste(getwd(), "/Data", sep = "")

zipped_data.filename <- "conways-reverse-game-of-life-2020.zip"
zipped_data.path <- file.path(data.dir.path, zipped_data.filename)

unzip(zipfile = zipped_data.path, exdir = data.dir.path)

train_data.filename <- "train.csv"
train_data.path <- file.path(data.dir.path, train_data.filename)
train_data <- read_csv(train_data.path)

test_data.filename <- "test.csv"
test_data.path <- file.path(data.dir.path, test_data.filename)
test_data <- read_csv(test_data.path)

rm(Packages, p, data.dir.path, test_data.filename, test_data.path,
  train_data.filename, train_data.path, zipped_data.filename, zipped_data.path)
```

2.2 Brute-Force Approach

The `aggregated_solve_S_start()` function previously defined represents our algorithm to solve Conway's GoL using brute force. Although the "true" brute-force approach of trial and error would involve trying every possible combination of 0s and 1s for the starting grid's cells and then evolve the grid forward the specified number of time-steps to check if the resulting grid coincides with the input descendent grid, our approach differs in three ways, allowing a significant (but far from sufficient) improvement in runtimes: it makes use of The

GoL's rules to check whether or not each candidate grid ancestor satisfies the mandatory conditions imposed on any grids that are to legitimately evolve into the given descendent grids, it eliminates the need to guess the values of certain cells in the ancestor grids, and it incorporates parallel processing to take advantage of all of my computer's logical processors.

The first of these features improves runtimes by replacing the step of evolving each possible candidate grid ancestor forward via a time-consuming loop with many iterations with a simple vectorized evaluation of a logical expression, whereas the second feature cuts runtimes by simply reducing the maximum number of possible candidate grid ancestors that must be checked at each time step, and the third feature speeds up the algorithm by kicking it off on multiple threads simultaneously, with each execution of the algorithm cycling through the various cells of the ancestor grids in a different order in a sort of race.

We can justify replacing the step of evolving each candidate grid ancestor forward to check for identity with a vectorized condition evaluation by first noting that the rules of Conway's GoL can be expressed using propositional logic as:

$\langle\{S_t = 1\} \implies \{[N_t \neq 0] \wedge [(N_t \in \{3,4\}) \implies (S_{t+1} = 1)] \wedge [(N_t \notin \{3,4\}) \implies (S_{t+1} = 0)]\}\rangle \wedge \langle\{S_t = 0\} \implies \{[N_t \neq 9] \wedge [(N_t = 3) \implies (S_{t+1} = 1)] \wedge [(N_t \neq 3) \implies (S_{t+1} = 0)]\}\rangle$.

This statement can then be shown to be true if and only if the following statement, which serves as the expression that we evaluate for each candidate grid ancestor, its corresponding neighborhood sums and the input descendent grid of interest, is true:

$\langle\{S_{t+1} = 0\} \vee \{[N_t \in \{3,4\}] \wedge [(N_t \neq 4) \vee (S_t = 1)]\}\rangle \wedge \langle\{S_{t+1} = 1\} \vee \{[N_t \neq 3] \wedge [(N_t \neq 4) \vee (S_t = 0)]\}\rangle \wedge \langle\{[N_t \neq 0] \vee [S_t = 0]\} \wedge \{[N_t \neq 9] \vee [S_t = 1]\} \wedge \{S_t \in \{0,1\}\}\rangle$

As for the second feature of our brute-force approach, we simply observe that only those parent grid cells that happen to be located within a neighborhood of a cell that's living in the child grid need consideration, as all other parent grid cells can be assumed to be dead without contradicting the observed outcome of the child grid.

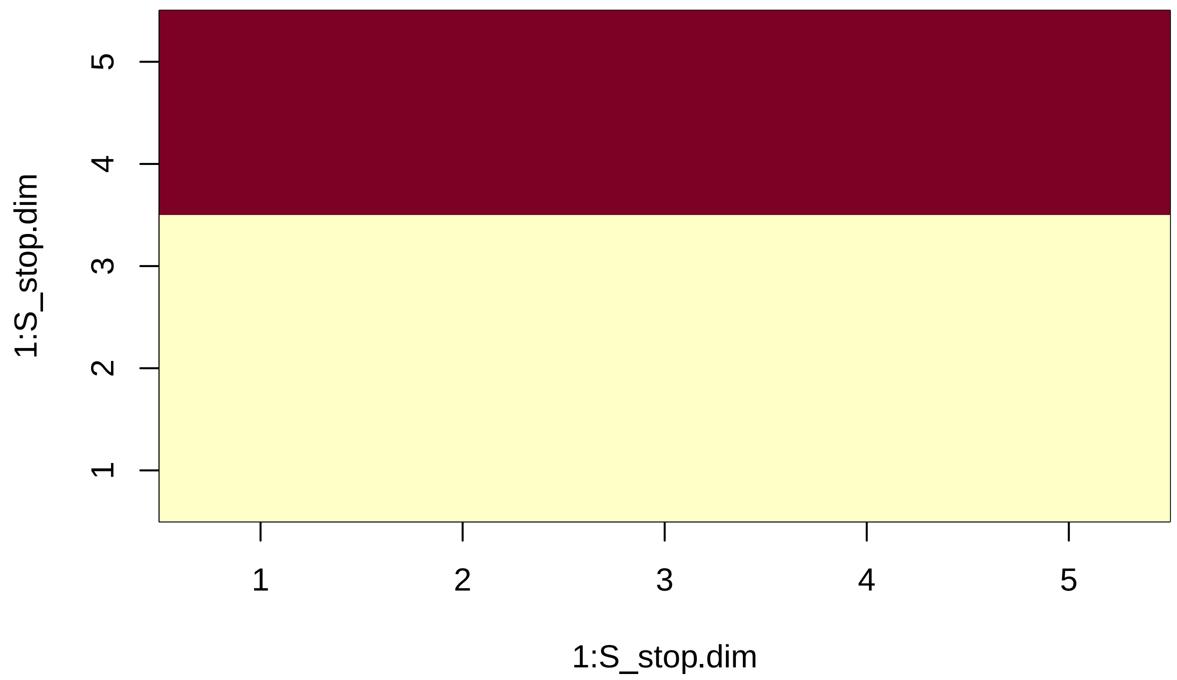
Said differently, only when such parent grid cells are tried as living can a candidate parent grid contradict the observed child grid after the values of the parent grid cells belonging to at least one living child grid cell's neighborhood have been determined appropriately. Rather than interpreting the dead state as separate and distinct from the living state, we can interpret the dead state as the absence of the living state, and focus solely on the child grid's living cells.

Lastly, our algorithm makes use of some of R's *parallel processing* packages and functions to leverage my computer's multiple logical processors. In essence, each of 20 threads kick off the same trial-and-error algorithm, but cycle through the ancestor grid cells in a different order while communicating with each other in the process, so that once the first thread has arrived at a solution, the others are notified and prompted to stop execution, and only this first thread's solution is returned by the algorithm.

The following code provides us with three examples of The GoL and shows the results and runtimes of our brute-force algorithm. We note that, using this algorithm, for each time step involved, there can be up to \$ 2^n \$ trial-and-error iterations, where \$ n \$ is the number of unique cells belonging to at least 1 of the descendent grid's living cells' neighborhoods.

Some simple arithmetic coupled with the produced runtimes and the realization that very few of the ending grids' cells in these examples are living will quickly reveal how attempting to solve 50,000 25x25 GoL grids using our brute-force approach would require far too much time to be of any significant value in practice (and our lifetimes).

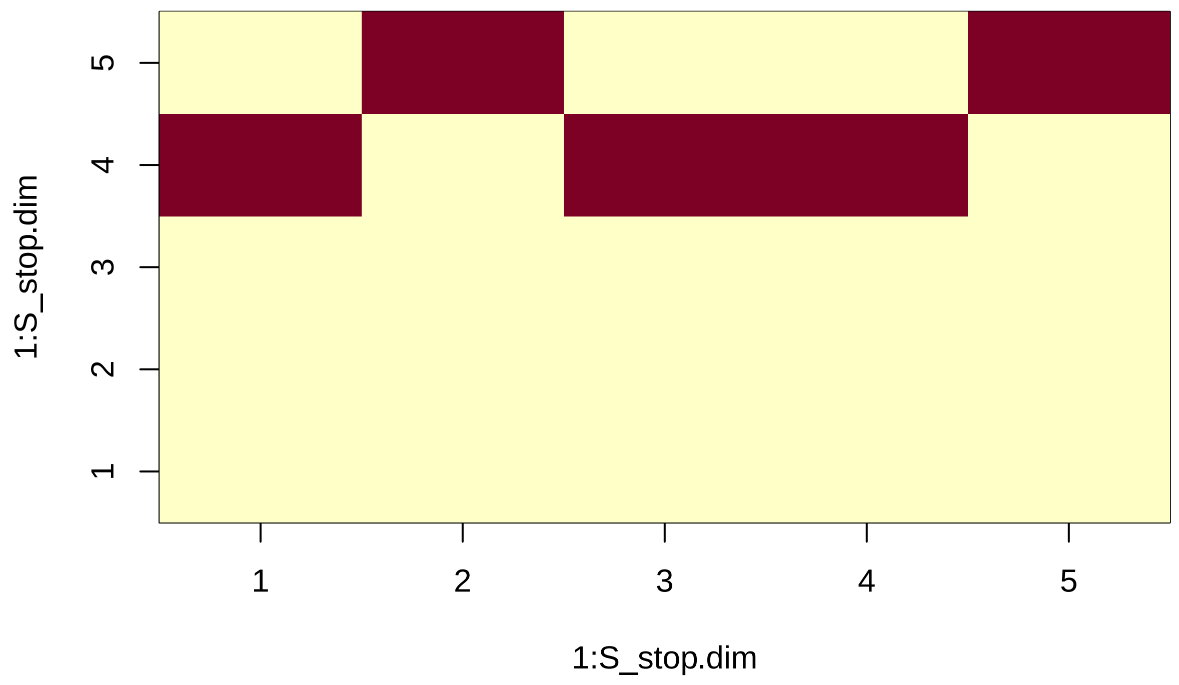
```r
# Example 1 (5x5 grid with delta = 1):

S_stop <-
  c(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

S_stop.dim <- sqrt(length(S_stop))

S_stop.mtrx <- matrix(S_stop, c(S_stop.dim, S_stop.dim), byrow = TRUE)

S_stop.image <- image(1:S_stop.dim, 1:S_stop.dim, t(S_stop.mtrx[S_stop.dim:1, ]))
```



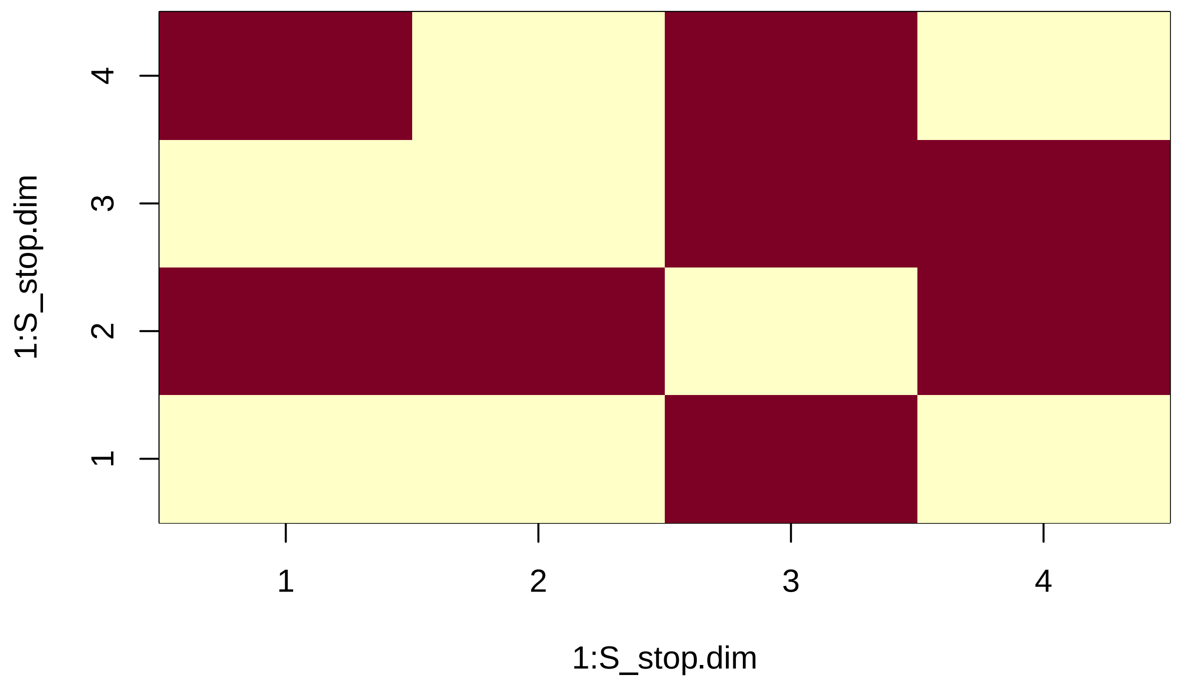
```r
delta <- 1

system.time(S_start_seq <- aggregated_solve_S_start(S_stop, delta))
```

```
##    user  system elapsed
##    0.16    0.06    1.19
```

```r
S_start.mtrx <- matrix(S_start_seq[[1]][["value"]],
  c(S_stop.dim, S_stop.dim), byrow = TRUE)

S_start.image <- image(1:S_stop.dim, 1:S_stop.dim,
  t(S_start.mtrx[S_stop.dim:1, ]))
```



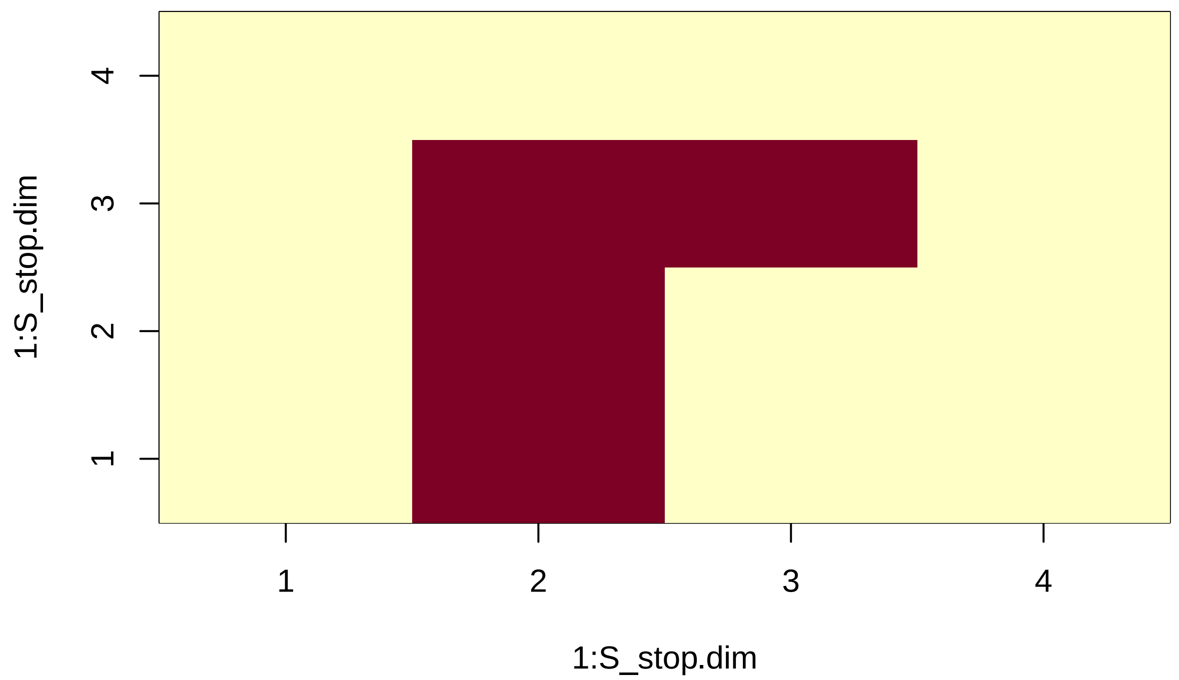
```r
# Example 2 (4x4 grid with delta = 5):

S_stop <- c(1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0)

S_stop.dim <- sqrt(length(S_stop))

S_stop.mtrx <- matrix(S_stop, c(S_stop.dim, S_stop.dim), byrow = TRUE)

S_stop.image <- image(1:S_stop.dim, 1:S_stop.dim, t(S_stop.mtrx[S_stop.dim:1, ]))
```

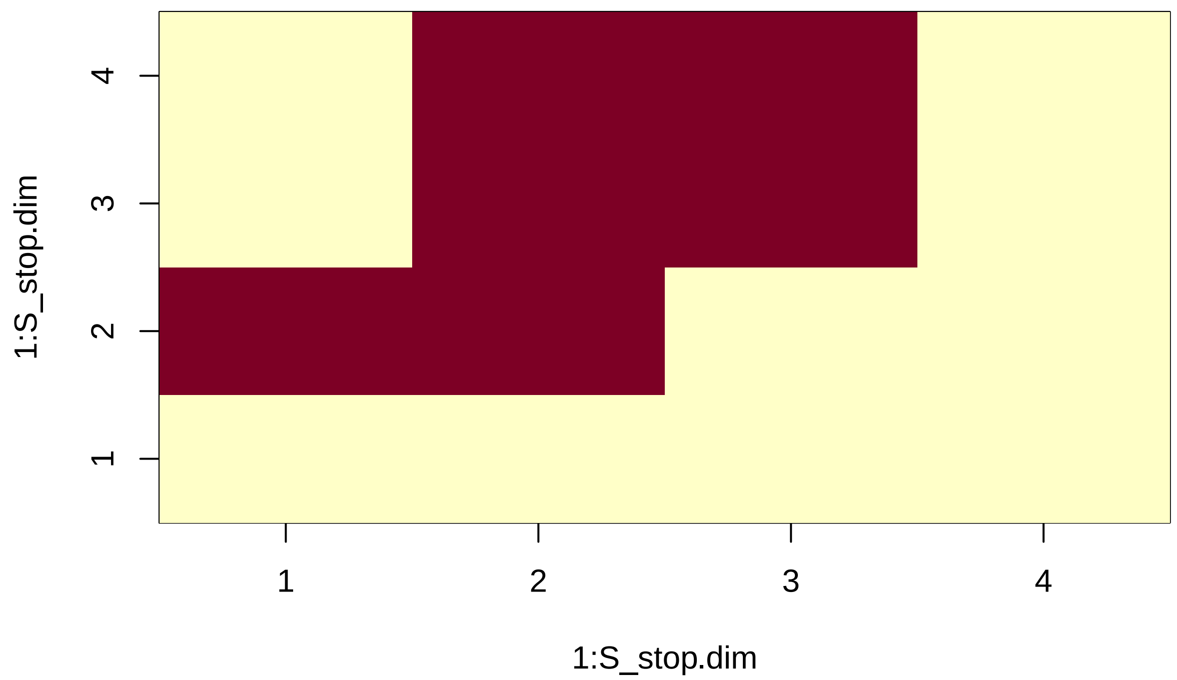


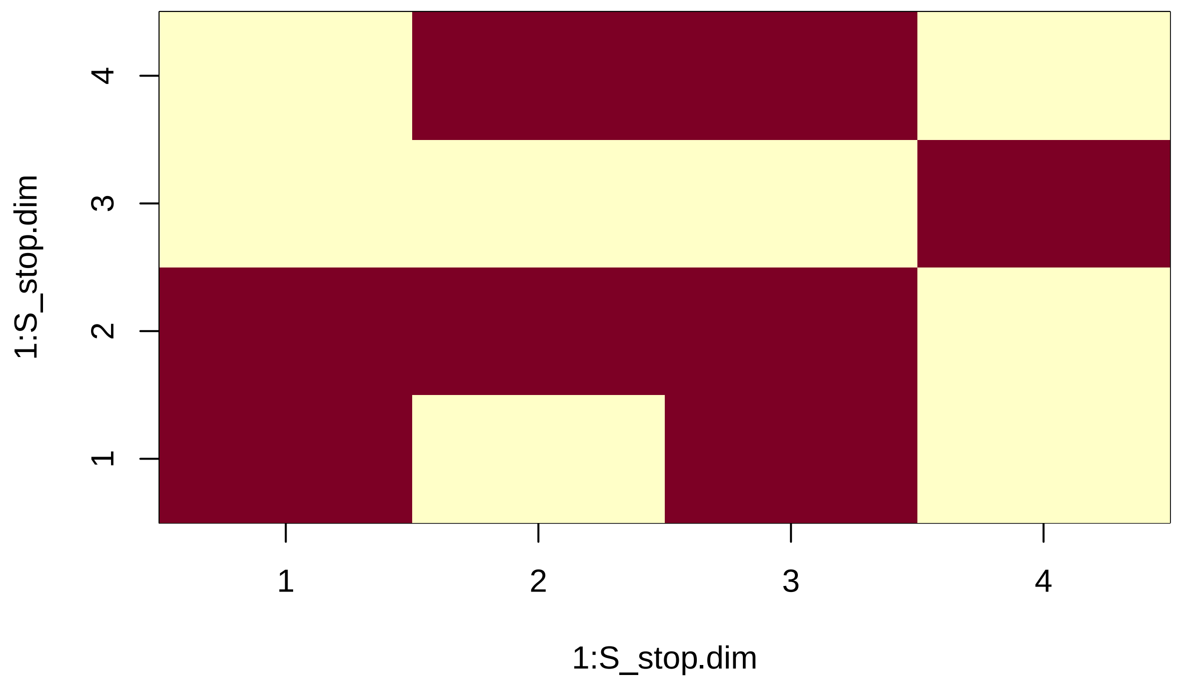
```
delta <- 5

system.time(S_start_seq <- aggregated_solve_S_start(S_stop, delta))
```

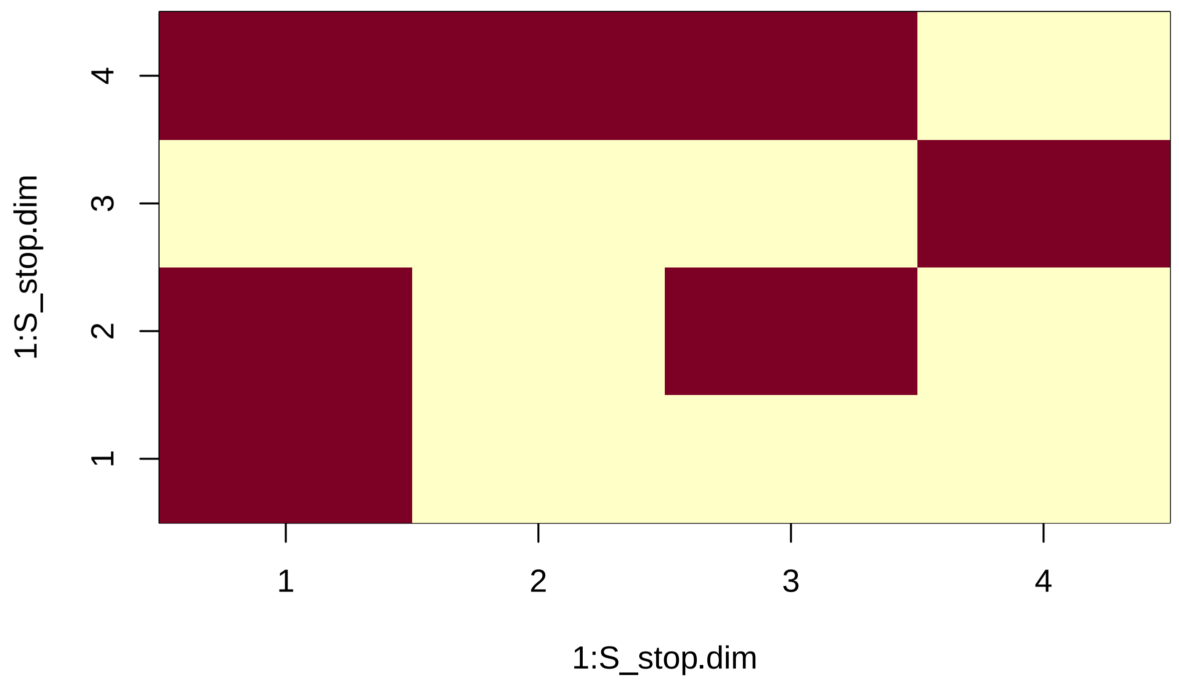
```
##    user  system elapsed
##    0.17    0.03   13.86
```

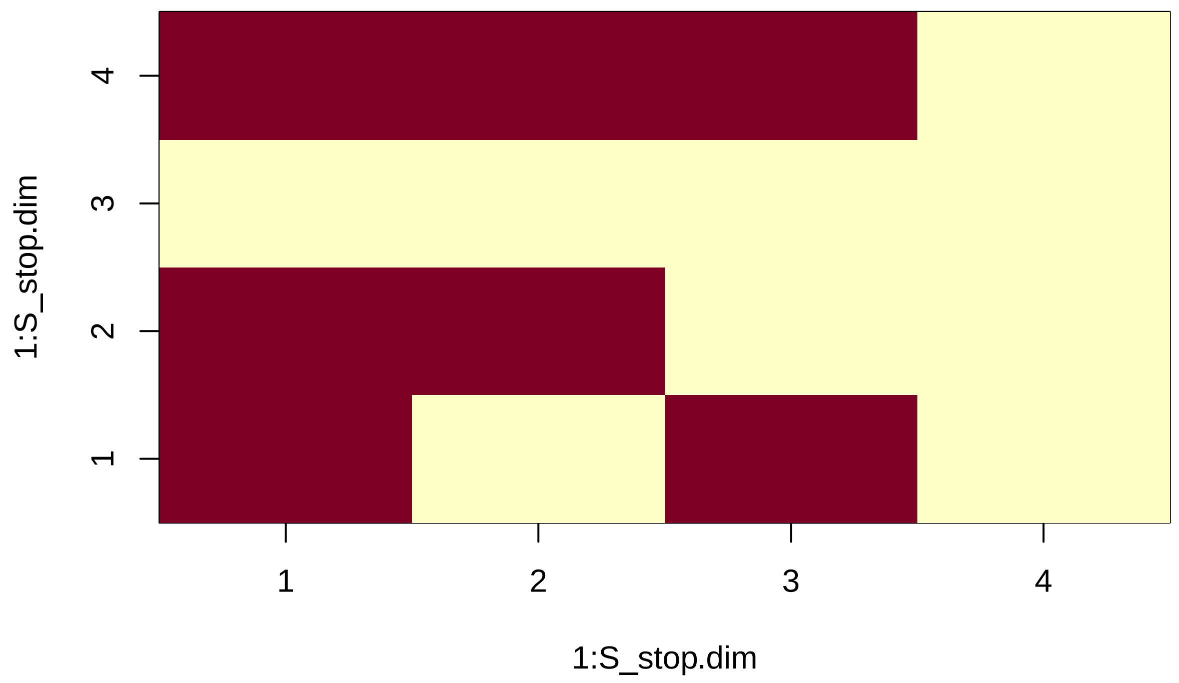
```
for(j in 1:delta){
  S_start.mtrx <- matrix(S_start_seq[[j]][["value"]],
    c(S_stop.dim, S_stop.dim), byrow = TRUE)
  S_start.image <- image(1:S_stop.dim, 1:S_stop.dim,
    t(S_start.mtrx[S_stop.dim:1, ]))
}
```


1:S_stop.dim
1:S_stop.dim
1
2
3
4
1
2
3
4

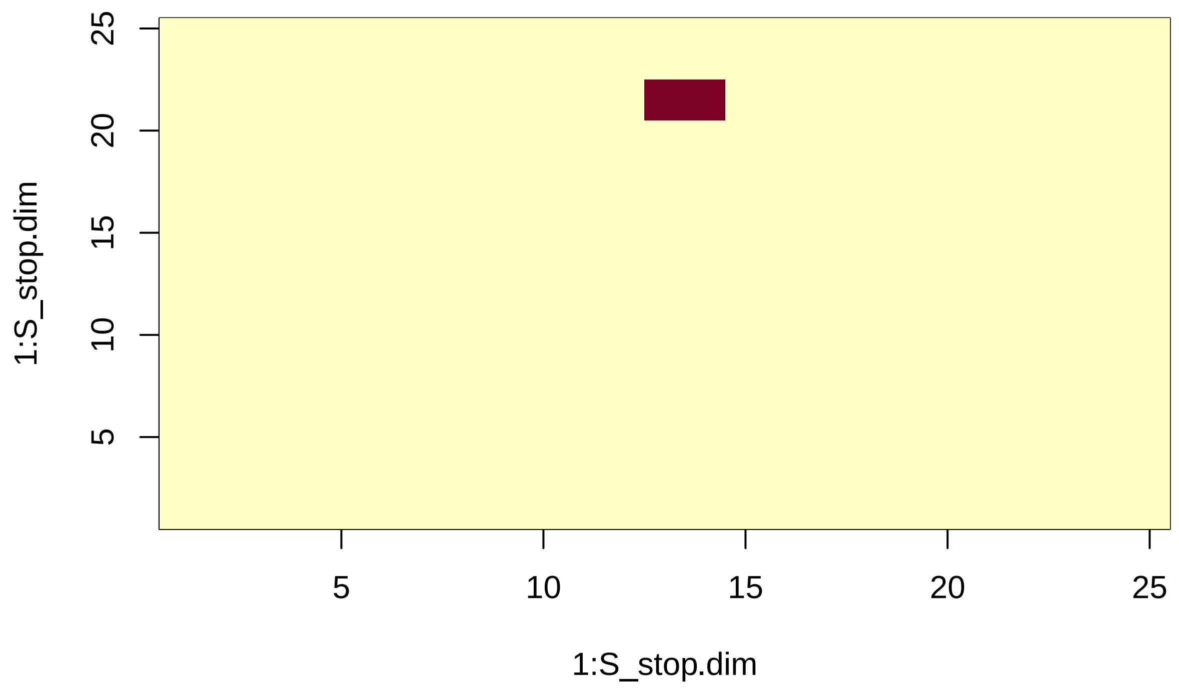

1:S_stop.dim
1
2
3
4
1
2
3
4
1:S_stop.dim


4
3
2
1
1:S_stop.dim
1
2
3
4
1:S_stop.dim


1:S_stop.dim
1
2
3
4
1
2
3
4
1:S_stop.dim



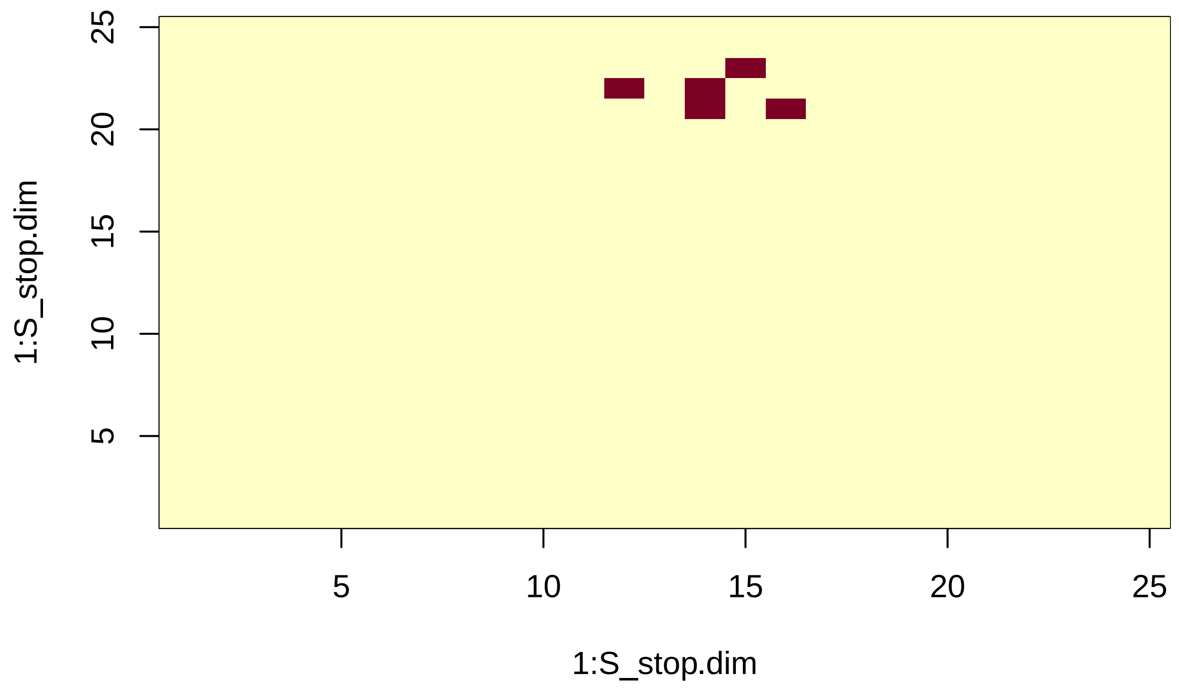
```r
# Example 3 (25x25 grid with delta = 3):

S_stop <-
  c(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

S_stop.dim <- sqrt(length(S_stop))

S_stop.mtrx <- matrix(S_stop, c(S_stop.dim, S_stop.dim), byrow = TRUE)

S_stop.image <- image(1:S_stop.dim, 1:S_stop.dim, t(S_stop.mtrx[S_stop.dim:1, ]))
```

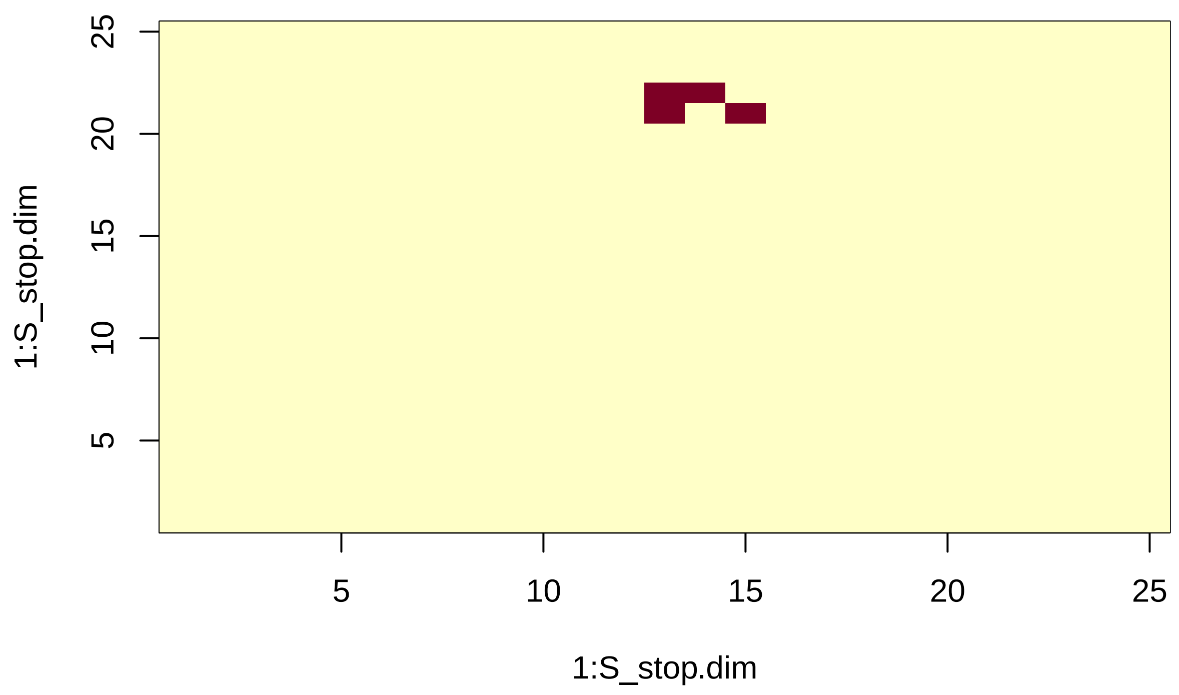


```
delta <- 3

system.time(S_start_seq <- aggregated_solve_S_start(S_stop, delta))
```

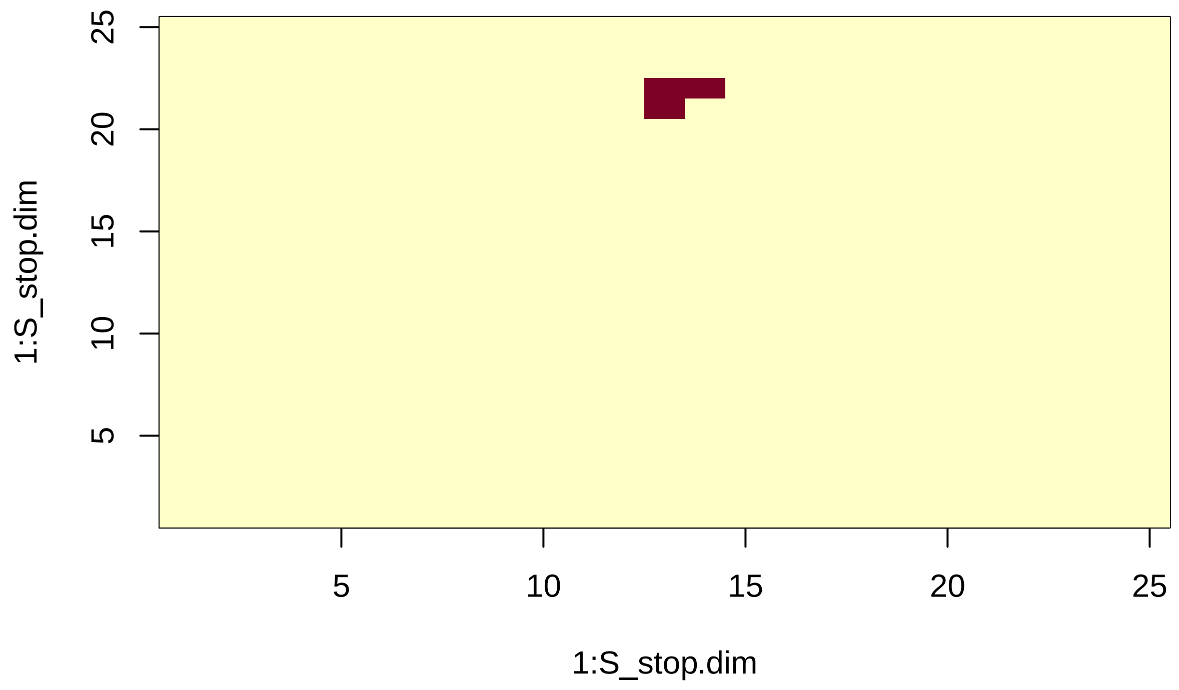
```
##    user  system elapsed
##    0.14    0.06    8.70
```

```r
for(j in 1:delta){
  S_start.mtrx <- matrix(S_start_seq[[j]][["value"]],
    c(S_stop.dim, S_stop.dim), byrow = TRUE)
  S_start.image <- image(1:S_stop.dim, 1:S_stop.dim,
    t(S_start.mtrx[S_stop.dim:1, ]))
}
```





1:S_stop.dim
1:S_stop.dim
5
10
15
20
25



```
stopCluster(reverse_GoL_cluster)

rm(S_start.mtrx, S_stop.mtrx, delta, j, S_start.image, S_stop, S_stop.dim,
  S_stop.image, reverse_GoL_cluster)
```

2.3 Machine Learning Methods

Given the realization provided by the previous examples that our brute-force approach can't be used in any meaningful way to predict the outcomes of the 50,000 25x25 grids in our test dataset, we now turn to applications of machine learning to remedy the computational complexity of Conway's GoL.

In what follows, we separately train two different machine learning models and combine their results to arrive at an *ensemble method* that we'll use to predict the outcomes of our test dataset's GoL configurations.

In particular, we'll be training a *Naive Bayes classifier*, followed by a *logistic regression model*, and after evaluating the performance of each over the test dataset, we'll combine these models and conclude our machine learning exposition by quickly evaluating an ensemble of the methods.

```r
# Train Naive Bayes model:

p_y_hat <- train_data %>% dplyr::summarize(across(start_0:start_624, mean))

S_stop.N.indices <- list()
p_x.y_hat <- list()
train_data.subset <- list()
X.test <- list()
p_y.x_hat.Bayes.test <- list()

for(k in 1:625){
  # As was done in function definition for define_aux_matrix(), determine indices
  # (i,j) and (I,J) relating index for each cell and indices for cells in its
  # neighborhood.

  j <- ifelse(k %% 25 == 0, 25, k %% 25)
  i <- ((k - j) + 25)/25
  J <- ifelse((j - 1):(j + 1) %% 25 == 0, 25, (j - 1):(j + 1) %% 25)
  I <- ifelse((i - 1):(i + 1) %% 25 == 0, 25, (i - 1):(i + 1) %% 25)

  # Define vector of indices for neighboring cells of kth cell.

  S_stop.N.indices[[k]] <- plyr::mdply(expand.grid(x = I, y = J),
    function(x, y){(x - 1)*25 + y}) %>% .[3] %>% pull()

  # Define subset of data corresponding to kth cell that includes the
  # observations of the kth starting grid cell value, delta, the kth stopping
  # grid cell value, and the corresponding neighborhood sum of cells in the
  # stopping grid.

  train_data.subset[[k]] <- train_data %>%
    dplyr::select(c(delta, 2 + k, 627 + S_stop.N.indices[[k]]))

  train_data.subset[[k]] <- train_data.subset[[k]] %>%
    mutate(X_3 = train_data.subset[[k]] %>% as.matrix() %>% .[, 3:11] %>%
    rowSums()) %>% dplyr::select(c(2, 1, 7, 12)) %>%
    setNames(c("Y", "X_1", "X_2", names(.)[4]))

  p_x.y_hat[[k]] <- list()

  # Given above defined subset of data pertaining to kth cell, define PMFs for
  # X|Y = 0 and X|Y = 1.

  for(m in 0:1){
    p_x.y_hat[[k]][[paste(m)]] <- train_data.subset[[k]] %>% filter(Y == m) %>%
      dplyr::select(-Y) %>% group_by(X_1, X_2, X_3) %>%
      summarize(p = n()/nrow(.), .groups = "drop") %>% arrange(X_1, X_2, X_3)
  }

  # Define subset of predictors pertaining to kth starting grid cell.

  X.test[[k]] <- test_data %>% dplyr::select(c(delta, 2 + S_stop.N.indices[[k]]))

  X.test[[k]] <- X.test[[k]] %>% mutate(X_3 = X.test[[k]] %>% as.matrix() %>%
    .[, 2:10] %>% rowSums()) %>% dplyr::select(c(1, 6, 11)) %>%
    setNames(c("X_1", "X_2", names(.)[3]))

  # Define PMF values for Y|X = x for starting grid cells in test dataset.

  p_y.x_hat.Bayes.test[[k]] <- p_y_hat[[k]]*ifelse(is.na(X.test[[k]] %>%
    left_join(p_x.y_hat[[k]][["1"]], by = c("X_1", "X_2", "X_3")) %>% .$p), 0,
    X.test[[k]] %>%
    left_join(p_x.y_hat[[k]][["1"]], by = c("X_1", "X_2", "X_3")) %>% .$p)/
    (p_y_hat[[k]]*ifelse(is.na(X.test[[k]] %>%
    left_join(p_x.y_hat[[k]][["1"]], by = c("X_1", "X_2", "X_3")) %>% .$p), 0,
    X.test[[k]] %>%
    left_join(p_x.y_hat[[k]][["1"]], by = c("X_1", "X_2", "X_3")) %>% .$p) +
    (1 - p_y_hat[[k]])*ifelse(is.na(X.test[[k]] %>%
    left_join(p_x.y_hat[[k]][["0"]], by = c("X_1", "X_2", "X_3")) %>% .$p), 0,
    X.test[[k]] %>%
    left_join(p_x.y_hat[[k]][["0"]], by = c("X_1", "X_2", "X_3")) %>% .$p))

  p_y.x_hat.Bayes.test[[k]] <-
    ifelse(is.na(p_y.x_hat.Bayes.test[[k]]), 0, p_y.x_hat.Bayes.test[[k]])
}

p_y.x_hat.Bayes.test <- suppressMessages(p_y.x_hat.Bayes.test %>% bind_cols() %>%
  as.matrix())

colnames(p_y.x_hat.Bayes.test) <- paste("p_y_", 1:625, ".x_hat.Bayes", sep = "")

Y_hat.Bayes.test <- ifelse(p_y.x_hat.Bayes.test >= 0.5, 1, 0)

colnames(Y_hat.Bayes.test) <- paste("Y_", 1:625, "_hat.Bayes", sep = "")

# Evolve predicted starting grids using Naive Bayes model the appropriate number
# of time steps to compare resulting grids to provided stopping grids.
```

```r
Y_hat.Bayes.test.evolved <-
  suppressMessages(bind_cols(test_data[2], Y_hat.Bayes.test) %>%
  apply(1, function(x){generate_S_stop(x[-1], x[1])}) %>% t())

# Evaluate Bayes model on test data:

Bayes_model_MAE <- mean(abs((test_data %>% dplyr::select(-c(1:2)) %>%
  as.matrix()) - Y_hat.Bayes.test.evolved))

# Train logistic regression models:

p_y.x_hat.logistic.test <- list()

# Define logistic regression model for each unique starting grid cell and
# predictions for test data.

for(k in 1:625){
  glm_temp <- train_data.subset[[k]] %>%
    glm(Y ~ X_1 + X_2 + X_3, data = ., family = "binomial")

  p_y.x_hat.logistic.test[[k]] <-
    predict(glm_temp, X.test[[k]], type = "response")
}

p_y.x_hat.logistic.test <-
  suppressMessages(p_y.x_hat.logistic.test %>% bind_cols() %>% as.matrix())

colnames(p_y.x_hat.logistic.test) <-
  paste("p_y_", 1:625, ".x_hat.logistic", sep = "")

Y_hat.logistic.test <- ifelse(p_y.x_hat.logistic.test >= 0.5, 1, 0)

colnames(Y_hat.logistic.test) <- paste("Y_", 1:625, "_hat.logit", sep = "")

# Evolve predicted starting grids using collection of logistic regression models
# the appropriate number of time steps to compare resulting grids to provided
# stopping grids.

Y_hat.logistic.test.evolved <-
  suppressMessages(bind_cols(test_data[2], Y_hat.logistic.test) %>%
  apply(1, function(x){generate_S_stop(x[-1], x[1])}) %>% t())

# Evaluate logistic regression models on test data:
```

```r
logistic_model_MAE <- mean(abs((test_data %>% dplyr::select(-c(1:2)) %>%
  as.matrix()) - Y_hat.logistic.test.evolved))

# Define and evaluate ensemble of Naive Bayes model and logistic model that uses
# the average of the conditional probabilities predicted by both models for its
# decision rule:

p_y.x_hat.ensemble.test <- (p_y.x_hat.Bayes.test + p_y.x_hat.logistic.test)/2

Y_hat.ensemble.test <- ifelse(p_y.x_hat.ensemble.test >= 0.5, 1, 0)

colnames(Y_hat.ensemble.test) <- paste("Y_", 1:625, "_hat.ensemble", sep = "")

Y_hat.ensemble.test.evolved <-
  suppressMessages(bind_cols(test_data[2], Y_hat.ensemble.test) %>%
  apply(1, function(x){generate_S_stop(x[-1], x[1])}) %>% t())

ensemble_model_MAE <- mean(abs((test_data %>% dplyr::select(-c(1:2)) %>%
  as.matrix()) - Y_hat.ensemble.test.evolved))

rm(i, I, j, J, k, m, glm_temp)
```

Chapter 3

Results

Running the above code, we can see that our Naive Bayes model results in a *mean absolute error (MAE)* of roughly 0.1421, whereas our logistic regression models result in an MAE of roughly 0.1396. Furthermore, our ensemble model results in an MAE of about 0.1395, a very slight improvement over our logistic regression models. Note that if we had randomly guessed at each cell's value, we would have expected an MAE of around 0.5.

Speaking loosely, we can summarize these results by saying that our machine learning models were roughly 86% accurate over the test dataset, despite taking hundredths of thousandths, if not millionths, of the amount of time it would have taken us to accomplish this same task using a trial-and-error approach. It wouldn't be an exaggeration to claim that we've just estimated 50,000 25x25 GoL grids with varying deltas using machine learning methods in less time than it would take us to solve a single well-populated 25x25 GoL grid over a single time step using our previously defined brute-force algorithm.

The 14% drop in accuracy here is hugely incommensurate to the time we've saved, demonstrating the power of machine learning as a solution to computationally complex problems.

Chapter 4

Conclusion

4.1 Summary

This report illustrates one of the more likely to be overlooked reasons why machine learning is of great importance: some problems are just too computationally demanding to address otherwise.

4.2 Limitations

With that being said, it's worth noting that we could have improved both our brute-force approach as well as our machine learning models quite a bit, although much more so for our machine learning models than for our brute-force algorithm.

Even so, we could have improved our brute-force algorithm in a few ways. For instance, we could have included code to ensure that the random permutation of the indices of the non-trivial cells assigned to each thread is unique - although different seeds are set for each of the threads, this doesn't guarantee that each thread is assigned a unique order in which it cycles through the non-trivial ancestor grid cells. Alternatively, we could have assigned each thread a particular deterministic trajectory through the grids. Moreover, we could have used probability theory to determine which cells our algorithm cycles through first, treating each cell as a Bernoulli variable, and each neighborhood sum as a binomial variable. Additionally, we could have built more logic into our algorithm by exploiting the relationships between neighboring neighborhood sums: as an example, each neighborhood sum shares 6 cells in common with the neighborhood sums with centers immediately to the north, south, east and west of its center, and shares 4 cells in common with those neighborhood sums with centers diagonally adjacent to its center.

For our machine learning models, we could have chosen more efficient algorithms for this type of problem. As an example, we could have used *k-nearest neighbors.* We also could have improved both models by expanding the predictive cells pertaining to each starting grid's cell for observations with delta > 1 by recursively including the cells surrounding each preceding grid's cells, resulting in either 9, 25, 49, 81 or 121 cells in each descendent grid containing some level of information about any given cell in the corresponding ancestral grid.

4.3 Future Work

Cellular automata theory is currently an active area of study among computer scientists and will continue to be at the forefront of theoretical computer science for the foreseeable future.