BROGLE Boris, PONTHON Rodolphe

L027: Report project:
Matrices and cellular automata

Table of contents

Introduction

The goal of this project was to provide a set of functions that can be used in the manipulation of an abstract data type called "Matrix".

In order to fulfil that goal, we had to provide a library in C language and also a ready-to-use program enabling the user to test the set of functions described below.

The purpose of this document is to describe the different data structures used, the algorithm of the main functions and the difficulties that we have met.

I. Used data types

To represent each Matrix, we used 4 different data types.

First, there is the Matrix data type, which contains the number of columns and rows, and a link to the first column and the first row.

```
typedef struct Matrix
{
	int colCount;
	int  rowCount;
	colElement* cols;
	rowElement* rows;
} Matrix;
```

Then, for each columns and rows in a given matrix, there is respectively a colElement and a rowElement. Each of these types has an index, a link to the next and the previous column, and also a link to the first cellElement of the given row/column.

```
typedef struct colElement
{
	int colN;
	cellElement* col;
	struct colElement* prevCol;
	struct colElement* nextCol;
} colElement;

typedef struct rowElement
{
	int rowN;
	cellElement* row;
	struct rowElement* prevRow;
	struct rowElement* nextRow;
} rowElement;
```

Each cellElement has its own value and its own row and col index. But it also has a pointer to the next cell element on the row and on the column where it is.

```
typedef struct cellElement
{
        int rowIndex;
        int colIndex;
        _Bool value;
        struct cellElement* nextCol;
        struct cellElement* nextRow;
} cellElement;
```

II. Main functions

1) newMatrix

The function newMatrix calls the function createAndAllocateMatrix at first, which creates and allocates a new matrix, and fill it with 0. After that, newMatrix just needs to browse the matrix that have just been created and browse the double dimension array, in order to replace each 0 of the matrix by the true value, contained in the array. In order to use dynamic allocation, we choose to use a double pointer on the two dimensions array.

```
Function newMatrix(table : **boolean, rows : integer, cols : integer) : Matrix
Begin
|       matrix : Matrix <- createAndAllocateMatrix(rows, cols)
|       if (cols(matrix) = UNDEFINED) then
|       |       newMatrix <- matrix
|       end if
|       i : integer <- 0
|       j : integer <- 0
|       rowElem : rowElement <- rows(matrix)
|       cellElem : cellElement
|       for i from 0 to rows do
|       |       cellElem <- row(rowElem)
|       |       for j from 0 to cols do
|       |       |       value(cellElem) <- table[i][j]
|       |       |       cellElem <- nextCol(cellElem)
|       |       end
|       |       rowElem <- nextRow(rowElem)
|       end
|       newMatrix <- matrix
End
```

The function createAndAllocateMatrix initializes each component of the structure Matrix, beginning with colCount and rowCount. Then, it creates the first column by creating every rowElement and the first cellElement's column, plus the links between them. A second loop builds other columns, and uses quite the same code as the first part that initialized cellElement. We also create the links between the different cellElements and colElements by having a pointer on the precedent element.

```
Function createAndAllocateMatrix(rows : integer, cols : integer) : Matrix
Begin
|       i : integer <- 0
|       j : integer <- 0
|       matrix <- createMatrix()
|       colCount(matrix) <- cols
|       rowCount(matrix) <- rows
|
|       if (rows < 1 OR cols < 1) then
|       |       cols(matrix) <- UNDEFINED        //if the matrix is empty, we just
|       |       rows(matrix) <- UNDEFINED        initialize the structure Matrix
|       else
|       |       //Creation of the first colElement
|       |       firstColElement <- createcolElement()
|       |       cols(matrix) <- firstColElement
|       |       prevCol(firstColElement) <- UNDEFINED
|       |       colN(firstColElement) <- 0
|       |       nextCol(firstColElement) <- UNDEFINED
|       |       //Creation of the first rowElement
|       |       firstRowElement : rowElement <- createRowElement()
|       |       rows(matrix) <- firstRowElement
|       |
|       |       prevRow(firstRowElement) <- UNDEFINED
|       |       rowN(firstRowElement) <- 0
|       |       nextRow(firstRowElement) <- UNDEFINED
|       |
|       |       secRowElement : rowElement
|       |
|       |       newCell : cellElement
|       |       prevCell : cellElement
|       |
|       |       for i from 0 to rows do
|       |       |       //Creation of the first col
|       |       |       newCell <- createCellElement()
|       |       |       rowIndex(newCell) <- i
|       |       |       colIndex(newCell) <- 0
|       |       |       nextCol(newCell <- UNDEFINED
|       |       |       nextRow(newCell <- UNDEFINED
|       |       |       value(newCell) <- 0          //We put 0-values in each cells
|       |       |       if (i ≠ 0) then
|       |       |       |       nextRow(prevCell) <- newCell
|       |       |       else
|       |       |       |       col(firstColElement) <- newCell
|       |       |       end if
|       |       |
|       |       |       prevCell <- newCell
|       |       |
|       |       |       //Linking the rowElements with every cellElements
|       |       |       row(firstRowElement) <- newCell
|       |       |       if (i+1 < rows) then
|       |       |       |       secRowElement <- createRowElement()
|       |       |       |       prevRow(secRowElement) <- firstRowElement
|       |       |       |       nextRow(firstRowElement) <- secRowElement
|       |       |       |       rowN(secRowElement) <- i+1
|       |       |       |       nextRow(secRowElement) <- UNDEFINED
|       |       |       |       firstRowElement <- secRowElement
|       |       |       end if
|       |       end
|       |
|       |       firstCells : cellElement
|       |       secColElement : colElement
|       |
|       |       for j from 1 to cols do
|       |       |
```

```
|       |       |       //Creation of the other cells and linking them with the
|       |       |       previous col cells'
|       |       |       secColElement <- createColElement()
|       |       |       prevCol(secColElement) <- firstColElement
|       |       |       nextCol(firstColElement) <- secColElement
|       |       |       colN(secColElement) <- j
|       |       |       nextCol(secColElement) <- UNDEFINED
|       |       |
|       |       |       firstCells <- col(firstColElement)
|       |       |       for i from 0 to rows do
|       |       |       |       newCell <- createCellElement()
|       |       |       |       rowIndex(newCell) <- i
|       |       |       |       colIndex(newCell) <- j
|       |       |       |       nextCol(newCell) <- UNDEFINED
|       |       |       |       nextRow(newCell) <- UNDEFINED
|       |       |       |       value(newCell) <- 0 //We put 0-values in each cells
|       |       |       |       if (i ≠ 0) then
|       |       |       |       |       nextRow(prevCell) <- newCell
|       |       |       |       else
|       |       |       |       |       col(secColElement) <- newCell
|       |       |       |       end if
|       |       |       |
|       |       |       |       prevCell <- newCell
|       |       |       |       nextCol(firstCells) <- newCell
|       |       |       |       if (i+1 < rows) then
|       |       |       |       |       firstCells <- nextRow(firstCells)
|       |       |       |       end if
|       |       |       end
|       |       |       firstColElement <- secColElement
|       |       end
|       end if
|       createAndAllocateMatrix <- matrix
End
```

2) PrintMatrix

For the printMatrix function, we chose to replace the 0-values with a “□” and the 1-values with a “■” so it is easier to see for the user.

The function is a simple transversal of the matrix. When we detect a 1-value, we print a “■” and a “□“ otherwise.

```
Function printMatrix(m : Matrix) : Void
Begin
|       if (isMatrixEmpty(m)) then
|       |       print(“Empty matrix!”)
|       else
|       |       rowElem : rowElement
|       |       rowElem <- rows(m)
|       |       cellElem : cellElement
|       |
|       |       while (rowElem ≠ UNDEFINED) do
|       |       |       cellElem <- row(rowElem)
|       |       |
|       |       |       while(cellElem ≠ UNDEFINED)
|       |       |       |       if (value(cellElem) = 1)
|       |       |       |       |       print(“■ ”)
```

```
|       |       |       |       else
|       |       |       |       |       print("□ ")
|       |       |       |       end if
|       |       |       |
|       |       |       |       cellElem <- nextCol(cellElem)
|       |       |       end
|       |       |       rowElem <- nextRow(rowElem)
|       |       end
|       end if
End
```

3) isMatrixEmpty, isColumnEmpty, isMatrixSquare

The function isMatrixEmpty detects if there are no colElement in the input Matrix, if so, we consider the matrix as empty and we return true.

```
Function isMatrixEmpty(m : Matrix) : Boolean
Begin
|       if (cols(m) = UNDEFINED) then
|       |       isMatrixEmpty <- TRUE
|       end if
|       isMatrixEmpty <- FALSE
End
```

The function isMatrixSquare just compares the number of rows and columns to check if it is the same, and return true is so.

```
Function isMatrixSquare(m : Matrix) : Boolean
Begin
|       if (colCount(m) = rowCount(m))
|       |       isMatrixSquare <- TRUE
|       end if
|       isMatrixSquare <- FALSE
End
```

The function isColumnEmpty allow the user to check if a column of the matrix is empty, which means that the index of the column doesn't exist. The function isRowEmpty works exactly on the same way, that's why we are going to write the algorithm of isColumnEmpty only. These functions return true if the matrix is empty and false if they aren't.

```
Function isColumnEmpty(m : Matrix, nCol : integer) : boolean
Begin
|       if (isMatrixEmpty(m)) then
|       |       isColumnEmpty <- TRUE
|       end if
|       if (nCol < 0) then
|       |       isColumnEmpty <- TRUE
|       end if
|       if (nCol >= colCount(m)) then
|       |       isColumnEmpty <- TRUE
|       end if
|       mColeElem : colElement
|       mColElem <- cols(m)
|
```

```
|       while (colN(mColElem) < nCol) do          //We browse the columns to reach the
|       |       mColElem <- nextCol(mColElem)     column that the user wants to check
|       end
|       if (colN(mColElem) = nCol)                //If the transversal colElement has
|       |       isColumnEmpty <- FALSE            the same index than the user's, it
|       end if                                    means that the column exists
|       isColumnEmpty <- TRUE                     If it is greater than the user's index
End                                               it means that the column doesn't exist
```

4) equalsMatrix

The equalsMatrix function is returning true when the two input matrix are identical, that includes when they are both empty, and false otherwise.

For that, we do a simultaneous transversal of both the matrix if and only if they have both the same size.

If it is the case, we just check out if all the values from the first matrix are the same in the second one, and if at one moment we find a difference between them, we return directly false.

```
Function equalsMatrix(m1 : Matrix, m2 : Matrix) : Boolean
Begin
|       m1Empty : Boolean <- isMatrixEmpty(m1)
|       m2Empty : Boolean <- isMatrixEmpty(m2)
|
|       if (m1Empty AND m2Empty) then
|       |       equalsMatrix <- TRUE
|       else    if(m1Empty OR m2Empty)
|       |       |       equalsMatrix <- FALSE
|       |       end if
|       else    if ((colCount(m1)=colCount(m2)) AND (rowCount(m1)=rowCount(m2))) then
|       |       |       m1RowElem : rowElement
|       |       |       m1RowElem <- rows(m1)
|       |       |       m2RowElem : rowElement
|       |       |       m2RowElem <- rows(m2)
|       |       |
|       |       |       m1CellElem : cellElement
|       |       |       m2CellElem : cellElement
|       |       |
|       |       |       while (m1RowElem ≠ UNDEFINED) do        // Transversal of the 2
|       |       |       |       m1CellElem <- row(m1RowElem)       matrix
|       |       |       |       m2CellElem <- row(m2RowElem)
|       |       |       |
|       |       |       |       while(m1CellElem ≠ UNDEFINED) do
|       |       |       |       |       if((value(m1CellElem)) ≠ (value(m2CellElem))) then
|       |       |       |       |       |       equalsMatrix <- FALSE
|       |       |       |       |       end if
|       |       |       |       |       m1CellElem <- nextCol(m1CellElem)
|       |       |       |       |       m2CellElem <- nextCol(m2CellElem)
|       |       |       |       end
|       |       |       |       m1RowElem <- nextRow(m1RowElem)
|       |       |       |       m2RowElem <- nextRow(m2RowElem)
|       |       |       end
|       |       |       equalsMatrix <- TRUE
```

```
|       |       end if
|       end if
|       equalsMatrix <- FALSE
End
```

5) sumMatrix

We voluntary chose not to explain the mulMatrix function, indeed, it is the same as the sumMatrix function, with just a different operator.

In the sumMatrix function, we chose to return a new matrix, which is created and allocated inside the function.

After some tests and after we did that, we do the simultaneous transversal of the 2 input matrix and the new matrix just created before, and we store into this matrix the result of the sum (OR operation) in this matrix.

```
Function sumMatrix(m1 : Matrix, m2 : Matrix) : Matrix
Begin
|       m1Empty : Boolean <- isMatrixEmpty(m1)
|       m2Empty : Boolean <- isMatrixEmpty(m2)
|
|       if (m1Empty OR m2Empty) then
|       |       print("Impossible to compute the operation due to empty matrix")
|       |       sumMatrix <- m1
|       else    if((colCount(m1) = colCount(m2))AND(rowCount(m1) = rowCount(m2)))then
|       |       |       matrix : Matrix
|       |       |       matrix <- createAndAllocateMatrix(rowCount(m1), colCount(m1))
|       |       |
|       |       |       m1RowElem : rowElement     // Creation of the new matrix and
|       |       |       m1RowElem <- rows(m1)        the pointers to do the transversal
|       |       |       m2RowElem : rowElement         of the two other
|       |       |       m2RowElem <- rows(m2)
|       |       |       mRowElem : rowElement
|       |       |       mRowElem <- rows(matrix)
|       |       |
|       |       |       m1CellElem : cellElement
|       |       |       m2CellElem : cellElement
|       |       |       mCellElem : cellElement
|       |       |
|       |       |       while (m1RowElem ≠ UNDEFINED) do  // Going from a row to
|       |       |       |       m1CellElem <- row(m1RowElem)          another
|       |       |       |       m2CellElem <- row(m2RowElem)
|       |       |       |       mCellElem <- row(mRowElem)
|       |       |       |
|       |       |       |       while (m1CellElem ≠ UNDEFINED) do   //OR operation below
|       |       |       |       |       value(mCellElem) <- value(m1CellElem) OR
|       |       |       |       |                           value(m2CellElem)
|       |       |       |       |
|       |       |       |       |       m1CellElem <- nextCol(m1CellElem)   // Going from
|       |       |       |       |       m2CellElem <- nextCol(m2CellElem)      cols to cols
|       |       |       |       |       mCellElem <- nextCol(mCellElem)
|       |       |       |       end
|       |       |       |       m1RowElem <- nextRow(m1RowElem)
|       |       |       |       m2RowElem <- nextRow(m2RowElem)
|       |       |       |       mRowElem <- nextRow(mRowElem)
|       |       |       end
```

```
|       |       |       sumMatrix <- matrix
|       |       end if
|       end if
|       print("The two input matrix do not have the same size")
|       sumMatrix <- m1
End
```

6) andColSequenceOnMatrix

andColSequenceOnMatrix is quite the same function as orColSequenceOnMatrix, andRowSequenceOnMatrix and orRowSequenceOnMatrix : we just need to change the operator OR and AND and to change the way we travel the matrix (rows or columns). This function creates another matrix, the result of the application of an AND operator between each members of the columns of the initial matrix. The result matrix will have one less column than the initial one.

```
Function andColSequenceOnMatrix(m : Matrix) : Matrix
Begin
|       if (isMatrixEmpty(m)) then
|       |       print("Empty matrix, unable to compute the operation")
|       |       andColSequenceOnMatrix <- m
|       end if
|       colElem : colElement <- cols(m)
|
|       if (nextCol(colElem) = UNDEFINED) then
|       |       print("The matrix has not enough columns to compute the operation,
|       |       will be returned without any modification")
|       |       andColSequenceOnMatrix <- m
|       end if
|       //Creation of the new matrix that will be returned
|       newMatrix : Matrix <- createAndAllocateMatrix(rowCount(m), colCount(m)-1)
|       newColElem : colElement <- cols(newMatrix)
|       newCell : cellElement <- col(newColElem)
|       cell : cellElement
|
|       while (nextCol(colElem) ≠ UNDEFINED) do
|       |       cell <- col(colElem)
|       |       newCell <- col(newColElem)
|       |
|       |       while(cell ≠ UNDEFINED) do
|       |       |       value(newCell) <- value(cell) && value(nextCol(cell))
|       |       |       newCell <- nextRow(newCell)       //here is the operator && (AND)
|       |       |       cell <- nextRow(cell)             that can be replaced by the
|       |       end                                       operator || (OR)
|       |       colElem <- nextCol(colElem)
|       |       newColElem <- nextCol(newColElem)
|       end
|       andColSequenceOnMatrix <- newMatrix
End
```

7) applyRule

For the function applyRule, we choose to use two sub-functions.

The first is a function called xorMatrix, which computes the XOR operation between two matrixes.

The second, called matRule, is a function which is able to do a fundamental rule on an input matrix, for instance, when we call this function with the rule 2, it translate the input matrix on the left.

The applyRule function works that way:

First, we decompose the input Rule into a succession of fundamental rule, then, for the first fundamental rule, we just translate the matrix. If there are more rules, we know it is a composed rule, so we compute the XOR operation between the input matrix translated with a fundamental rule, and the precedent result matrix.

As an example, for the application of Rule 6, we start the decomposition in binary, we then start with rule 2, that translate the matrix on the left, and then we compute the XOR operation between the precedent matrix (rule 2), and another matrix translated by rule 4. And that returns the same as if we did manually all the XOR in the matrix between the right cells and the bottom-right cells.

```
Function applyRule(m : Matrix, ruleID : integer, times : integer) : Matrix
Begin
|       // We first create two temporary matrixes and copy the input Matrix into m1.
|       m1 : Matrix <- createAndAllocateMatrix(rowCount(m),colCount(m))
|       m2 : Matrix <- createAndAllocateMatrix(rowCount(m),colCount(m))
|       m1 <- copyMatrix(m1,m)
|
|       if (ruleID < 512 AND ruleID > 0) then
|       |       k : integer <- 0
|       |       for k from 0 to times do
|       |       |       m <- copyMatrix(m,m1)
|       |       |       result : integer <- ruleID
|       |       |       n : integer <- 0
|       |       |       i : integer <- 0
|       |       |       while result ≠ 0 do // Binary decomposition
|       |       |       |       if ((result % 2) = 1) then
|       |       |       |       |       i <- i + 1
|       |       |       |       |       if (i = 1) then   // The first fundamental rule is
|       |       |       |       |       |                 just a translation
|       |       |       |       |       |       m1 <- matRule(m1,power(2,n))
|       |       |       |       |       else              // For each more rule
|       |       |       |       |       |       m2 <- copyMatrix(m2,m)
|       |       |       |       |       |       m2 <- matRule(m2,power(2,n))
|       |       |       |       |       |       m1 <- xorMatrix(m1,m2)
|       |       |       |       |       end if
|       |       |       |       end if
|       |       |       |       result <- result/2
|       |       |       |       n <- n + 1
|       |       |       end
|       |       end
|       else
|       |       print("Wrong Rule id")
```

```
|       end if
|       freeMatrix(m2)
|       m <- copyMatrix(m,m1)
|       freeMatrix(m1)
|       applyRule <- m
End
```

Below is the function that computes the XOR operation between two input Matrix.
It works like the functions that compute the sum or the multiplication between two given matrix but with a difference: the result matrix is not a new one, but is put into m1.

```
Function xorMatrix(m1 : Matrix, m2 : Matrix) : Matrix
Begin
|       m1Empty : boolean <- isMatrixEmpty(m1)
|       m2Empty : boolean <- isMatrixEmpty(m2)
|
|       if (m1Empty OR m2Empty) then
|       |       print("Impossible to compute due to empty matrix")
|       |       xorMatrix <- m1
|       else    if((colCount(m1)=colCount(m2)) AND (rowCount(m1)=rowCount(m2))) then
|       |       |       m1RowElem : rowElement
|       |       |       m1RowElem <- rows(m1)
|       |       |       m2RowElem : rowElement
|       |       |       m2RowElem <- rows(m2)
|       |       |       m1CellElem : cellElement
|       |       |       m2CellElem : cellElement
|       |       |
|       |       |       while (m1RowElem ≠ UNDEFINED) do         // We compute the XOR
|       |       |       |       m1CellElem <- row(m1RowElem)        operation for each
|       |       |       |       m2CellElem <- row(m2RowElem)        cells.
|       |       |       |
|       |       |       |       while(m1CellElem ≠ UNDEFINED) do
|       |       |       |       |       value(m1CellElem) <- (value(m1CellElem) AND
|       |       |       |       |        not(value(m2CellElem)))OR(not(value(m1CellElem))  |
|       |       |       |       |        AND value(m2CellElem)
|       |       |       |       |       m1CellElem <- nextCol(m1CellElem)
|       |       |       |       |       m2CellElem <- nextCol(m2CellElem)
|       |       |       |       end
|       |       |       |
|       |       |       |       m1RowElem <- nextRow(m1RowElem)
|       |       |       |       m2RowElem <- nextRow(m2RowElem)
|       |       |       end
|       |       |
|       |       |       xorMatrix <- m1
|       |       end if
|       end if
|       printf("Error! The two input matrix have not the same size")
|       xorMatrix <- m1
End
```

The matRule function is basically just a simple function that chose the right translation to do, depending of the fundamental rule given in input.

```
Function matRule(m : Matrix, ruleID : integer) : Matrix
Begin
|       switch(ruleID)
|       |       case 2
|       |       |       m <- rule2(m)
|       |       break
|       |
|       |       case 4
|       |       |       m <- rule2(m)
|       |       |       m <- rule8(m)
|       |       break
|       |
|       |       case 8
|       |       |       m <- rule8(m)
|       |       break
|       |
|       |       case 16
|       |       |       m <- rule8(m)
|       |       |       m <- rule32(m)
|       |       break
|       |
|       |       case 32
|       |       |       m <- rule32(m)
|       |       break
|       |
|       |       case 64
|       |       |       m <- rule128(m)
|       |       |       m <- rule32(m)
|       |       break
|       |
|       |       case 128
|       |       |       m <- rule128(m)
|       |       break
|       |
|       |       case 256
|       |       |       m <- rule2(m)
|       |       |       m <- rule128(m)
|       |       break
|       end
|       matRule <- m
End
```

Below is one of the 4 functions used to translate depending of the fundamental Rule.
Here is the rule2 function which is translating to the left the matrix values. First, it translates all the values inside the matrix to the left, and then, it fills the last column with 0.
The 3 other functions are working the same way with some minor changes for the rule 64 and 128. (See C code)

```
Function rule2(m : Matrix): Matrix
Begin
|       if (isMatrixEmpty(m)) then
|       |       rule2 <- m
|       end if
|
|       if (colCount(m) ≠ 1) then          // If there is only 1 column, we replace it
|       |       colElem : colElement          with a 0-filled column by changing the
|       |       colElem <- cols(m)            values.
|       |       cellElem : cellElement
|       |       cellElem <- col(colElem)
|       |
|       |       while(cellElem ≠ UNDEFINED) do
```

```
|       |       |       value(cellElem) <- 0
|       |       |       cellElem <- nextRow(cellElem)
|       |       end
|       else
|       |       colElem : colElement
|       |       colElem <- cols(m)
|       |       nextColElem : colElement
|       |       nextColElem <- nextCol(colElem)
|       |       cellElem : cellElement
|       |       nextCellElem : cellElement
|       |
|       |       while (nextColElem ≠ UNDEFINED) do      // We put the value of the
|       |       |       cellElem <- col(colElem)           right column in the left one
|       |       |       nextCellElem <- col(nextColElem)
|       |       |
|       |       |       while(cellElem ≠ UNDEFINED) do
|       |       |       |       value(cellElem) <- value(nextCellElem)
|       |       |       |       cellElem <- nextRow(cellElem)
|       |       |       |       nextCellElem <- nextRow(nextCellElem)
|       |       |       end
|       |       |       colElem <- nextCol(colElem)
|       |       |       nextColElem <- nextCol(nextColElem)
|       |       end
|       |
|       |       cellElem <- col(colElem)                // We fill the last column with
|       |       while(cellElem ≠ UNDEFINED) do             0 values.
|       |       |       value(cellElem) <- 0
|       |       |       cellElem <- nextRow(cellElem)
|       |       end
|       end if
|       rule2 <- m
End
```

III. Encountered difficulties

The main difficulty that we have met was about the applyRule function. Indeed, we knew that it was too difficult to compute this function by doing the different XOR operations directly in the input Matrix.

And so we had to think about how to do this by another way. It took a while to find that we had to compute the XOR operation between the translated matrix and the precedent matrix (see applyRule section for more details) for the composed rules.

We also had some difficulties to debug some point of the project but that was more or less always quickly done.

Finally, we had some trouble to find how to do the library in the compilation, we did not see it in the labs and it was not clear enough on the web.

IV. Conclusion

This project has enabled us to work in pair and to train us with linked list. We worked regularly on it and helped us eachother.

It was also quite interessant to see the applyRule function working and what is at the origin of the image manipulation in computer science.

But we didn't implement the alternative way to represent the 0-filled columns or rows, this is what is left to do.

We also tried to make our code as optimized as we could, but it is certainly a better way for doing some of our functions.