UTBM LO27 A16

Matrices and Cellular Automata

Project Report

Yann Le Chevanton & William Veal Phan

02/01/2016

Table of contents

1. Introduction

The aim of this project was the implementation of a dynamic representation of Boolean matrices and the function to manipulate them. Among the functions implemented, we had to implement a function allowing the manipulation of Matrices as cellular automaton. Furthermore, by considering our matrices as black and white bitmap pictures, we could see practical applications of Mathematics in the field of image manipulation. We also had an optional condition that we chose to consider which was to implement our matrices as sparse matrices. This choice influenced in a very radical way how we had to think our functions and how we designed them.

In this report, we will start with a description of the various datatypes we implemented, and in a second part we will continue with the description of the functions used to manipulate our data-types and explanations of our choices of algorithmic design. We will then conclude with a short and as objective as possible analysis of our work.

2. Abstract data-types implemented

2.1. Matrix

As asked in the project, we implemented a Matrix data-type as a structure containing its number of columns, its number of rows, and pointers to doubly linked list representing its columns and rows. This structure is essentially the same as the one described in the project descriptive, and the only changes the sparse Matrix representation brings to this abstract data-type is the behavior of the columns, cells and rows.

$$Matrix = colCount \times rowCount \times cols \times rows$$

2.1.1. rowElement and colElement

Both our rowElement and colElement have the same overall structure: they are both composed of their row or column number, a pointer to the next element in the list, another to the previous and a pointer to the first cellElement of their row or column. The main difference with the type described in the project comes from the

sparse nature of our Matrix: we chose not to create empty columns or rows containing only zeros, which means the indices of the elements in the lists can be discontinuous. And, as seen latter in this document, not having a complete list of elements indexed from 1 to a fixed integer will seriously impact the way we think the behavior of even the simplest task.

$$rowElement = rowN \times nextRow \times prevRow \times row$$
$$colElement = colN \times nextCol \times prevCol \times col$$

2.1.2. cellElement

Our cellElement is a structure representing the cells of our Matrix. It is composed of a row and a column index, and pointers to the next element in the row and another toward the next element in a column. The main difference with the project descriptive is resulting from a mistake from us: we mistook nextCol for the next in the column rather than the element in the next column (same for nextRow), and decided to keep it this way.

Because we were representing sparse matrices, we dimmed unnecessary the value field: an existing cellElement can only represent a Boolean value TRUE (or 1).

$$cellElement = colIndex \times rowIndex \times nextCol \times nextRow$$

2.2. Points and listPoints

To facilitate the creation of our matrices we decided to introduce another data-type: Points and listPoints, which is a list of Points as a pointer to its first element. A Points represents TRUE in the Matrix it will build. It is composed of a column index x and a row index y. We admit it is not the most intuitive naming convention, but we started with this and didn't want to change it afterward. It also possesses a pointer to the next Points in the listPoints. A listPoints is essentially the list of non-zero cells of a Matrix.

$$Points = x \times y \times nextP$$

The only function we implemented for this data-type is insertTailPoints which insert a new Points at the end of a list of Points given an argument by pointer.

2.3. listMatrix

To obtain a Matrix we also need to add the number of rows and the number of columns to our abstract data-type. We then introduce the listMatrix datatype. A listMatrix is composed of a listPoints, and an integer n giving the number of rows and another integer p giving the number of columns.

$$listMatrix = list \times n \times p$$

We also wanted to implement an array representation of a Matrix, but due to time consuming difficulties encountered during the debugging phase of our program we decided to drop the idea. The premises are still available as comments in our source code.

3. Functions implemented

3.1. insertRow and insertCol

These functions are used to insert row or columns in a Matrix. We thought it woul be useful to explain them to understand the structure of our Matrix matrices. Both function having the same behavior, we will only detail insertRow.

```
Lexicon :
newel : the new row we want to insert
tmp : a temporary variable
m : address of a Matrix in which we will insert a row
index : the index of the ne row to insert

function insertRow(m : Matrix*, index : integer) : Matrix*
BEGIN
newel, tmp : rowElement*
newel <- NULL
tmp <- rows(m)
if tmp = NULL
then
       rowN(newel) <- index
       rows(m) <- newel
       prevRow(newel) <- NULL
       nextRow(newel) <- NULL
       row(newel) <- NULL
else
       if rowN(tmp) > index
       then
              rowN(newel) <- index
              nextRow(newel) <- tmp
              prevRow(tmp) <- newel
              rows(m) <- newel
                                                    prevRow(newel) <- NULL
```

```
                row(newel) <- NULL
        else
                while not(isEmpty(nextRow(tmp))) AND index > rowN(nextRow(tmp)) do
                        tmp<-nextRow(tmp)
                done
                if index != rowN(tmp)
                then
                        if nextRow(tmp) = NULL
                        then
                                rowN(newel) <- index
                                nextRow(newel) <- NULL
                                prevRow(newel) <- tmp
                                row(newel) <- NULL
                                nextRow(tmp) <- newel
                        else
                                rowN(newel)<- index
                                nextRow(newel) <- nextRow(tmp)
                                prevRow(newel) <- tmp
                                row(newel) <- NULL
                                nextRow(tmp) <- newel
                                prevRow(nextRow(newel)) <- newel
                        end if
                end if
        end if
end if
insertRow <- m
END
```

3.2. removeRow and removeCol

These functions are used to remove a row or a column from a Matrix. Both functions have the same behavior.

These functions are a prime example of a simple task becoming a very complex one due to the sparse nature of our matrices, each time we removed a column or a row, we had to check if its removal and the subsequent removal of its cells would imply the removal of other rows or columns.

3.3. NewMatrix

This function is the one we use to create a new matrix as a Matrix from a listMatrix. Its general behavior is to traverse the list of the listMatrix given as argument and initialize the column in which the new cell should be if it doesn't exist yet, insert the new cell in the column, create the row if it doesn't yet exist and link the cells of this row to the new element. We also do not forget to free the elements as we advance in our list.

```
Lexicon :
pt : adress of a Points we will ad to our Matrix
prToFree : address of a Points we use to free our memory spaces
currCol : temporary column address
currCell : temporary address of a cell
allocate() : function allocating memory space

function newMatrix(m : listMatrix*) : Matrix*
BEGIN
pt, ptToFree : Points*
currCol : colElement*
currRow : rowElement*
currCell, newCell : cellElement*
pt <- list(m)
newMat : Matrix* <- allocate(Matrix*)
colCount(newMat) <- p(m)
rowCount(newMat) <- n(m)
cols(newMat) <- NULL
rows(newMat) <- NULL

if isEmpty(pt)
then
	newMatrix <- newMat
end if

while not(isEmpty(pt)) do

	currCol <- cols(newMat)
	currRow <- rows(newMat)

	while not(isEmpty(currCol)) AND colN(currCol) < x(pt) do
		currCol <- nextCol(currCol)
	done

	if(isEmpty(currCol) OR colN(currCol) > x(pt))
	then
		newMat <- insertCol(newMat, x(pt))
		currCol <- cols(newMat)
		while colN(currCol != x(pt)) do
			currCol <- nextCol(currCol)
		done
	end if

	newCell <- allocate(cellElement*)
	colIndex(newCell) <- x(pt)
	rowIndex(newCell) <- y(pt)
	nextRow(newCell) <- NULL
	nextCol(newCell) <- NULL

	if isEmpty(col(currCol))
	then
		col(CurrCol) <- newCell
	else
		currCell <- col(currCol)
		if rowIndex(currCell > rowIndex(newCell))
		then
			nextCol(newCell) <- currCell
			col(currCol) <- newCell
		else
			while(nextCol(currCell) != NULL AND rowIndex(nextCol(currCell)) <
rowIndex(newCell)) do
				currCell <- nextCol(currCell)
			done
			if(nextCol(currCell) = NULL)
```

```
                        then
                                nextCol(currCell) <- newCell
                        else
                                nextCol(newCell) <- nextCol(currCell)
                                nextCol(currCell) <- newCell
                        end if
                end if
        end if

        while currRow != NULL AND rowN(currRow) < rowIndex(newCell) do
                currRow <- nextRow(currRow)
        done
        if currRow = NULL OR rown(currRow > rowIndex)
        then
                newMat <- insertRow(newtMat, rowIndex(newCell))
                currRow <- rows(newMat)
                while(rowN(currRow != rowIndex(newCell))) do
                        currRow <- nextRow(currRow)
                done
        end if
        currCell <- row(currRow)
        if currCell = NULL
        then
                row(currRow) <- newCell
        else
                if colIndex(currCell) > colIndex(newCell)
                then
                        nextRow(newCell) <- currCell
                        row(currRow) <- newCell
                else
                        while(nextRow(currCell != NULL AND colIndex(nextRow(currCell)) <
colINdex(newCell))) do
                                currCell <- nextRow(currCell)
                        done
                        if nextRow(currCell) = NULL
                        then
                                nextRow(currCell) <- currCell
                        else
                                nextRow(newCell) <- nextRow(currCell)
                                nextRow(currCell) <- newCell
                        end if
                end if
        end if
        ptToFree <- pt
        pt <- nextP(pt)
        free(ptToFree)
        list(m) = NULL
done
free(m)
newMatrix <- newMat
END
```

3.4. sumMatrix

The sumMatrix function compute the sum of two matrices given as arguments. Our matrices being Boolean matrices, this is equivalent to doing an OR operation between each cells of both matrices. The general idea here was to travers both matrices row by row. If the rowN of both rows are not coinciding, we know that we have to add all the cells of at least one of the two, arbitrarily the one with the smallest index, before

moving on. And if the indexes are the same, we traverse the rows and do the same with each cell, before moving both pointers.

```
Lexicon :
a, b : pointers to two Matrix we want to sum up
rowa, rowb : pointers of rowElement we use in the traversal of the matrices a and b
cella, cellb : pointers of cellElement we use in the traversal of rowa and rowb
newel : pointer to a listMatrix which we will use to create the Matrix corresponding to the
sum of a and b

function sumMatrix(a : Matrix*,b : Matrix*) : Matrix*
BEGIN
rowa, rowb : rowElement*
cella, cellb : cellElement*
newel : listMatrix*
if isMatrixSameSize(a,b) <- FALSE OR isMatrixEmpty(a) OR isMatrixEmpty(b)
then
       print("ERROR : wrong size")
       sumMatrix <- NULL
else
       rowa <- rows(a)
       cella <- NULL
       rowb <- rows(b)
       cellb <- NULL
       n(newel) <- rowCount(a)
       p(newel) <- colCount(b)
       list(newel) <- NULL
       while not(isEmpty(rowa)) AND not(isEmpty(rowb)) do
              cella <- row(rowa)
              cellb <- row(rowb)
              if(rowN(rowa) <- rowN(rowb))
              then
                     while not(isEmpty(cella)) AND not(isEmpty(cellb)) do
                            if colIndex(cella) > colIndex(cellb)
                            then
                                   list(newel) <- insertTailPoints(rowIndex(cellb),
colIndex(cellb), list(newel))
                                   cellb <- nextRow(cellb)
                            else
                                   if colIndex(cella) <- colIndex(cellb) then
                                          list(newel) <- insertTailPoints(rowIndex(cellb),
colIndex(cellb), list(newel))
                                          cellb <- nextRow(cellb)
                                          cella <- nextRow(cella)
                                   else
                                          list(newel) <- insertTailPoints(rowIndex(cella),
colIndex(cella), list(newel))
                                          cella <- nextRow(cella)
                                   end if
                            end if
                     done
                     while not(isEmpty(cella)) do
                            list(newel)<-insertTailPoints(rowIndex(cella), colIndex(cella),
list(newel))
                            cella <- nextRow(cella)
                     done
                     while not(isEmpty(cellb)) do
                            list(newel)<-insertTailPoints(rowIndex(cellb), colIndex(cellb),
list(newel))
                            cellb <- nextRow(cellb)
                     done
                     rowa <- nextRow(rowa)
                     rowb <- nextRow(rowb)
```

```
		else
			if rowN(rowa) < rowN(rowb)
			then
				while not(isEmpty(cella)) do
					list(newel) <- insertTailPoints(rowIndex(cella),
colIndex(cella), list(newel))
					cella <- nextRow(cella)
				done
				rowa <- nextRow(rowa)
			else
			while not(isEmpty(cellb)) do
				list(newel) <-insertTailPoints(rowIndex(cellb),
colIndex(cellb), list(newel))
				cellb <- nextRow(cellb)
			done
			rowb <- nextRow(rowb)
		end if
	end if
	while not(isEmpty(rowa)) do
		cella <- row(rowa)
		while not(isEmpty(cella)) do
			list(newel) <- insertTailPoints(rowIndex(cella), colIndex(cella),
list(newel))
			cella <- nextRow(cella)
		done
		rowa <- nextRow(rowa)
	done
	while not(isEmpty(rowb)) do
		cellb <- row(rowb)
		while not(isEmpty(cellb)) do
			list(newel) <-insertTailPoints(rowIndex(cellb), colIndex(cellb),
list(newel))
			cellb <- nextRow(cellb)
		done
		rowb <- nextRow(rowb)
	done
		sumMatrix <- newMatrix(newel)
end if
END
```

3.5. mulMatrix

The mulMatrix function compute the multiplication of two matrices given as arguments. This is equivalent to applying the rules of the multiplication of matrices by replacing multiplications with the AND operator. The behavior is almost the same as for sumMatrix, except that we need to have as many columns in the first Matrix as there are rows in the second so that we can iterate on the rows of the first Matrix and the columns of the second one. It is also simpler and faster because we need to consider adding Points to our listPoints iff our operands are non-zeros.

```
Lexicon :
A,B : pointers to two matrices we want to multiply. A must have as many columns as B has
rows
row : a pointer to a row we use in the traversal of our Matrix A
col : a pointer to a column we us in the traversal of B
rowCell : pointer to a cell we use in our traversal of row
colCell : poiinter to a cell we use in our traversal of col
```

```
newel : pointer to a listMatrix we will use to create the new Matrix corresponding to A x B

function mulMatrix(A : Matrix*, B : Matrix*) : Matrix*
BEGIN
row : rowElement*
col : colElement*
rowCell : cellElement*
colCell : cellElement*
newel : listMatrix*
row <- rows(A)
col <- cols(B)
rowCell <- row(rows(A))
colCell <- col(cols(B))
if A->colCount <- rowCount(B)
then
       n(newel) <- rowCount(A)
       p(newel) <- colCount(B)
       list(newel) <- NULL
       while not(isEmpty(row)) do
              while not(isEmpty(col)) do
                     rowCell <- row(row)
                     colCell <- col(col)
                     while not(isEmpty(rowCell)) AND not(isEmpty(colCell)) do
                            if colIndex(rowCell) = rowIndex(rowCell)
                            then
                                   list(newel) <-
insertTailPoints(rowN(row),rowN(row),list(newel))
                                   colCell <- NULL
                                   rowCell <- NULL
                            else
                                   if (colIndex(rowCell) > rowIndex(rowCell))
                                   then
                                          colCell <- nextCol(colCell)
                                   else

                                          rowCell <- nextRow(rowCell)
                                   end if
                            end if
                     done
                     col <- nextCol(col)
              done
              row <- row->nextRow
              col <- cols(B)
       done
else
       n(newel) <- 0
       p(newel) <- 0
       list(newel) <- NULL
end if
mulMatrix <- newMatrix(newel)
END
```

3.6. andColSequenceOnMatrix and andRowSequenceOnMatrix

These functions having the same overall behavior, we will only detail andColSequenceOnMatrix. This function creates a new Matrix with one less column. It results from the use of the AND operator between each column and its direct successor.

```
Lexicon :
m : pointer to the Matrix in which we operate
fcol, scol : pointers to columns in m. fcol is the left column in the computation
of the AND operation between fcol and scol
fcell, scell : pointer to cells in fcol and scol used in their traversal
newMat : pointer to the new listMatrix we use to build the result of our function

function andColSequenceOnMatrix(m : Matrix*) : Matrix*
BEGIN
fcol, scol : colElement*
fcell,scell : cellElement*
newMat : listMatrix*
if not(isMatrixEmpty(m))
then
	if not(isEmpty(cols(m))) AND not(isEmpty(nextCol(cols(m))))
	then
		fcol <- cols(m)
		scol <- nextCol(cols(m))
		fcell <- col(fcol)
		scell <- col(scol)
		n(newMat) <- rowCount(m)
		p(newMat) <- colCount(m) -1
		list(newMat) <- NULL
		while not(isEmpty(scol)) do
			if colN(scol) <- colN(fcol) + 1
			then
				fcell <- col(fcol)
				scell <- col(scol)
				while not(isEmpty(scell)) AND not(isEmpty(fcell)) do
					if rowIndex(fcell) <- rowIndex(scell)
					then
						list(newMat)                                <-
insertTailPoints(rowIndex(fcell), colIndex(fcell), list(newMat))
						fcell <- nextCol(fcell)
						scell <- nextCol(scell)
					else
						if rowIndex(fcell) < rowIndex(scell)
						then
							fcell <- nextCol(fcell)
						else
							if rowIndex(scell) < rowIndex(fcell)
							then
								scell <- nextCol(scell)
							end if
						end if
					end if
				done
			end if
			fcol <- nextCol(fcol)
			scol <- nextCol(scol)
		done
	else
		n(newMat) <- rowCount(m)
		p(newMat) <- colCount(m)
		list(newMat) <- NULL
	end if
else
	n(newMat) <- 0
```

```
	p(newMat) <- 0
	list(newMat) <- NULL
end if
andColSequenceOnMatrix <- newMatrix(newMat)
END
```

3.7. orColSequenceOnMatrix and orRowSequenceOnMatrix

Same as the former functions: we will only detail orColSequenceOnMatrix which compute the or operator between each column of the input Matrix and its direct successor.

```
Lexicon :
m : pointer to the Matrix in which we operate
fcol, scol : pointers to columns in m. fcol is the left column in the computation
of the OR operation between fcol and scol
fcell, scell : pointer to cells in fcol and scol used in their traversal
newMat : pointer to the new listMatrix we use to build the result of our function


function orColSequenceOnMatrix(m : Matrix*) : Matrix*
BEGIN
fcol, scol : colElement*
fcell,scell, tmp : cellElement*
newMat : listMatrix*
if not(isMatrixEmpty(m))
then
	if not(isEmpty(cols(m))) AND not(isEmpty(nextCol(cols(m)))
	then
		fcol <- cols(m)
		scol <- nextCol(cols(m))
		fcell <- col(fcol)
		scell <- col(scol)
		n(newMat) <- rowCount(m)
		p(newMat) <- colCount(m) -1
		while not(isEmpty(scol)) do
			if colN(scol) <- colN(fcol) + 1
			then
				fcell <- col(fcol)
				scell <- col(scol)
					while not(isEmpty(scell)) AND not(isEmpty(fcell)) do
						if rowIndex(fcell) <- rowIndex(scell)
						then
							list(newMat) <-
insertTailPoints(rowIndex(fcell), colIndex(fcell) , list(newMat))
							fcell <- nextCol(fcell)
							scell <- nextCol(scell)
						else
							if rowIndex(fcell) < rowIndex(scell)
							then
								list(newMat) <-
insertTailPoints(rowIndex(fcell), colIndex(fcell) , list(newMat))
								fcell <- nextCol(fcell)
							else
								if rowIndex(fcell) >
rowIndex(scell)
								then
									list(newMat) <-
insertTailPoints(rowIndex(scell), colIndex(scell)-1 , list(newMat))
```

```
                                                                scell <- nextCol(scell)
                                                        end if
                                                end if
                                        end if
                                done
                                while not(isEmpty(fcell)) do
                                        list(newMat) <-
insertTailPoints(rowIndex(fcell), colIndex(fcell) , list(newMat))
                                        fcell <- nextCol(fcell)
                                done
                                while not(isEmpty(scell)) do
                                        list(newMat) <-
insertTailPoints(rowIndex(scell), colIndex(scell)-1 , list(newMat))
                                        scell <- nextCol(scell)
                                done
                        else
                                while not(isEmpty(fcell))  do
                                        list(newMat) <-
insertTailPoints(rowIndex(fcell), colIndex(fcell) , list(newMat))
                                        fcell <- nextCol(fcell)
                                done
                        end if
                        fcol <- scol
                        scol <- nextCol(scol)
                done
        else
                if not(isEmpty(cols(m)) then
                        tmp <- col(cols(m))
                        n(newMat) <- rowCount(m)
                        p(newMat) <- colCount(m) -1
                        while not(isEmpty(tmp)) do
                                list(newMat) <- insertTailPoints(rowIndex(tmp),
colIndex(tmp) , list(newMat))
                                tmp <- nextCol(tmp)
                        done
                end if
        end if
else
        n(newMat) <- 0
        p(newMat) <- 0
        list(newMat) <- NULL
end if
orColSequenceOnMatrix <- newMatrix(newMat)
END
```

3.8. applyrules

This function take a Matrix, a rule identified by a number and an integer representing the number of times the rule will be applied as arguments. These rules correspond to manipulations on the input Matrix. Its behavior being complex to describe, we decided to explain it in further details below.

We decided to differentiate three cases: “the rule is complex”, “the rule is elementary” and “we cannot compute”. This impossibility an either result from our

rule being outside of the bounds of our set of rules or the Matrix given is either empty or full of FALSE cells.

applyRules updates the pointer m given as argument.

```
Lexicon :
m : pointer to the Matrix on which we operate
rule : integer identifying the rule to apply
times : integer indicating the number of times to apply the rule
i,k,l : integers, iteratives variables
currRow : pointer to a rowElement we use in the traversal of our Matrix
currCell : pointer to a cellElement used in the traversal of currRow
newMat : pointer to a listMatrix we will use to build our result.
listTrues : listPoints storing all the non-zero cells in the output Matrix
tempMat : pointer to a Matrix. We used it in case we had a segmentation fault

function applyRules(m : Matrix*, rule : integer, times : integer) : Matrix*
BEGIN
if(not(isMatrixEMpty(m)) AND rule >= 2 AND rule <= 511)
then
       i,k,l : integer
       dRule : array<BOOL>
       currRow : rowElement*
       currCell : cellElement*
       newMat : listMatrix*
       listTrues : listPoints
       tempMat : Matrix*
       for(i from 1 to times by +1) do
              switch(rule)
                     case 2 :
                            transLeft(m)
                     case 4 :
                            transUp(m)
                            transLeft(m)
                     case 8 :
                            transUp(m)
                     case 16 :
                            transRight(m)
                            transUp(m)
                     case 32 :
                            transRight(m)
                     case 64 :
                            transDown(m)
                            transRight(m)
                     case 128 :
                            transDown(m)
                     case 256 :
                            transleft(m)
                            transDown(m)
                     case else
                            newMat <- allocate(listMatrix)
                            lisTrues <- NULL
                            dRule <- decomposeRule(rule)
                            currRow <- rows(m)
                            n(newMat) <- rowCount(m)
                            p(newMat) <- colcount(m)
                            tempmat <- m
                            while currRow != NULL do
                                   currCell <- row(currRow)
                                   while(currCell != NULL) do

       if(applyRuleToCell(m,rowIndex(currCell),colIndex(currCell), dRule))
                                          then
```

```
                                                listTrues<-
insertTailPoints(rowIndex(currCell),colIndex(currCell), litTrues)
                                        end if
                                        for(k from -1 to 1 by +1) do
                                                for(l from -1 to 1 by +1) do

        if(isCellTrue(m,rowIndex(currCell)+k,colIndex(currCell)+l)      =      FALSE      AND
applyRuleToCell(m,rowIndex(currCell)+k,colIndex(currCell)+l)                              AND
containsPoints(colIndex(currCell)+l,rowIndex(currCell)+k,listTruues) = FALSE)
                                                                then
                                                                        listTrues<-
inserTailPoints(rowIndex(currCell)+k,colIndex(currCell)+l,lisTrues)
                                                                end if
                                                        done
                                                done
                                                currCell <- nextRow(currCell)
                                        done
                                        currRow <- nextRow(currRow)
                                done
                                list(newMat) <- listTrues
                                freeMatrix(tempMat)
                                m <- newMatrix(newMat)
                                newMat <- NULL
        done
end if
applyRules <- m
END
```

3.8.1. Elementary rules and translations functions

If the rule is an elementary rule, we decided we could consider its result as a simple translation of the Matrix. Thus, we introduced 4 translation procedures: transRight, transLeft, transUp, transDown. A diagonal translation is only the combination of 2 of those. Their behaviors being very similar we will only detail transRight.

```
Lexicon :
m : pointer to a Matrix we want to translate
currCol : pointer to a colElement we will use to iterate in our Matrix
currCell : pointer to a cell we use in the traversal of currCol

procedure transRight(m : Matrix)
BEGIN
currCol : colElement*
currCell : cellElement*
if(isMatrixEmpty(m) = FALSE)
        while(currCol != NULL) do
                if colN(currCol = colCount(m))
                then
                        currCol = NULL;
                        m <- removeCol(m,colCount(m));
                else
                        currCol <- nextCol(currCol)
                end if
        done
        currCol <- cols(m)
        while currCol != NULL do
                colN(currCol) <- colN(currCol) + 1
                currCell <- col(currCol)
```

```
            while(currCell != NULL) do
                colIndex(currCell) <- colIndex(currCell) + 1
                currCell <- nextCol<-currCell
            done
            currCol <- nextCol(currCol)
        done
end if
END
```

3.8.2. Complex rule decomposition

In the case of a complex rule, we decided to decompose the rule in its base-2 representation, and to keep them in the memory as a static array of 8 Booleans. We then use this array to apply the rule to each non-zero cell of the Matrix, and every zero-cells (not kept in the memory) in their vicinity.

```
Lexicon :
rule : the complex rule we want to decompose
decompose[9] : a 9 cells 1 dimensional array of booleans. It will contains the
decomposition of our rule
i : integer, iterative variable

function decomposeRule(rule : integer) : array<BOOL>
BEGIN
decompose[9] : array<BOOL>
i : integer
for(i from 8 to 0 by -1) do
        decompose[i] = rule / 2^i
        rule <- rule mod 2^i
done
decomposeRule <- decompose
END
```

3.8.3. applyRuleToCell

We considered that applying a rule to a Matrix is the same as applying the rul to each of its cells. We defined a function applyRuleToCell which purpose was to compute the successive XOR operations with the cells identified by the rule used.

```
Lexicon :
m : pointer to the Matrix in which is situated the cell
cellRow, cellCol : integers, give the position of the cell in the Matrix
dRule : an array of boolean containing the decomposition of the rule to apply. Obtained with
decomposeRule above
result : the result of our function. TRUE if the computation of every XOR operations indicates
the cell identified with cellRow and cellCol in the output Matrix of applyRules is TRUE,
FALSE otherwise

function applyRuleToCell(m : Matrix*, cellRow : integer, cellCol : integer, dRule :
array<BOOL>) : BOOL
BEGIN
result : BOOL <- FALSE

if(cellRow < 1 OR cellCol < 1 OR cellCol > colCoount(m) OR cellRow > rowCount(m))
```

```
then
       rapplyRuleToCell <- result
end if
if(dRule[0] = TRUE)
then
       result <- xor(result,isCellTrue(m,cellRow,cellCol))
end if
if(dRule[1] = TRUE)
then
       result <- xor(result,isCellTrue(m,cellRow,cellCol+1))
end if
if(dRule[2] = TRUE)
then
       result <- xor(result,isCellTrue(m,cellRow+1,cellCol+1))
end if

if(dRule[3] = TRUE)
then
       result <- xor(result,isCellTrue(m,cellRow+1,cellCol))
end if

if(dRule[4] = TRUE)
then
       result <- xor(result,isCellTrue(m,cellRow+1,cellCol-1))
end if

if(dRule[5] = TRUE)
then
       result <- xor(result,isCellTrue(m,cellRow,cellCol-1))
end if

if(dRule[6] = TRUE)
then
       result <- xor(result,isCellTrue(m,cellRow-1,cellCol-1))
end if

if(dRule[7] = TRUE)
then
       result <- xor(result,isCellTrue(m,cellRow-1,cellCol))
end if

if(dRule[8] = TRUE)
then
       result <- xor(result,isCellTrue(m,cellRow-1,cellCol+1))
end if
applyRuleToCell <- result
END
```

4. Conclusion

Finally, while working on this project we realized the pros and cons of having sparse matrices: on the one hand, it is less demanding in memory and a lot of operations will be much faster than with classical matrices. For example, deciding if a cell exist can be done by first looking at its index, then looking up its row or column before looking for the cell itself. On the other hand, implementing functions manipulating sparse matrices is much more complicated: we have more diverse situations, and it is

very easy to forget one case or do another mistake. And, as we realized while debugging, it can be very difficult to isolate the issue.

We also kept discovering new mistakes, bugs and glitches even as we were writing this report. As such, it is very likely that some are still remaining. We in particular think of memory leaks, as we found some and corrected them.