Data Structures & Algorithms

**Problem statement:**

Minesweeper is a [single-player](https://en.wikipedia.org/wiki/Single-player_video_game) [puzzle video game](https://en.wikipedia.org/wiki/Puzzle_video_game). The objective of the game is to clear a rectangular board containing hidden "[mines](https://en.wikipedia.org/wiki/Land_mine)” without detonating any of them, with help from clues about the number of neighboring mines in each field.

The game will be played through a console-based interface. The user will choose a size N for the board (>=5). The board will be randomly generated, with the density of the elements roughly equal to 25%. The NxN board will be presented with the unexplored spots marked with ‘-’ and a number of flags equal to the number of mines. The user can explore a spot or mark it with a flag. Whenever the user chooses to explore a spot he will either: lose the game if the spot is a mine or otherwise expand it such that he gains new information on the surrounding mines. The game ends when a mine is hit or when the there are no empty spots left on the board.

**Justification:**

The game is played on a rectangular board where the majority of the elements are 0 (the empty spots), thus the only relevant elements remain the spots occupied by a mine. This enables us to use a sparse matrix, since, by ignoring the empty spots, the data requires significantly less storage.

**ADT Domain specification & representation:**

SparseMatrix = { m | m is a container that represents a two-dimensional array, where each element has a unique position determined by 2 indexes (line & column) and a non-zero value. The elements are stored under the form triplet = (line, column, value), with { line ∈ Integer, column ∈ Integer, value ∈ TValue }.

The hashing will be done using the division method, while the collisions will be resolved by separate chaining.

For this particular problem perfect hashing could be achieved with a hash function that uses the line and column indexes; an example would be: lineIndex + columnIndex \* noColumns.

*TElem:*

line: Integer

column: Integer

value: TValue

*Node:*

next: ↑Node

element: TElem

*SparseMatrix:*

matrix: ↑Node[]

noLines: Integer

noColumns: Integer

hash: TFunction

size: Integer

**ADT Interface specification:**

*create(m, nol, noc)*

Pre: nol, noc ∈ Integer, nol > 0, noc > 0

Post: m ∈ SparseMatrix with nol lines and noc columns

*modify(m, l, c, v)*

Pre: m ∈ SparseMatrix; l, c ∈ Integer, v ∈ TValue

Post: m1 ∈ SparseMatrix; m1 = m where the TValue with key k on (l, c) = v

Throws: InvalidPositionException if l > m.noLines or c > m.noColumns

*lineNr(m)*

Pre: m ∈ SparseMatrix

Post: *lineNr* ∈ Integer, *lineNr* <- the number of lines in the matrix

*columnNr(m)*

Pre: m ∈ SparseMatrix

Post: *columnNr* ∈ Integer, *columnNr* <- the number of columns in the matrix

*getElement(m, l, c)*

Pre: m ∈ SparseMatrix; l,c ∈ Integer

Post: e ∈ TValue, e <- the value at (l, c) mapped with the key k

Throws: InvalidPositionException if l > m.noLines or c > m.noColumns

**ADT Implementation:**

Subalgorithm create(m, nol, noc):

for i <- 0, i < size, 1 do

m.matrix[i] <- NULL

m.noLines <- nol

m.noColumns <- noc

End-Subalgorithm

Complexity:

BC =AC = WC = ϴ(n), n = m.size – whenever a new matrix is created all the elements are always initialized with NULL

Subalgorithm modify(m, l, c, v)

if l > m.noLines or c > m.noColumns then

@Throw InvalidPositionException

end-if

k <- l + c \* m.noColumns

position <- m.hash(k)

node <- m.matrix[position]

if node = NIL and v = NIL then

return

else if node = NIL and v != NIL then

@allocate(newNode)

newNode.next <- NIL

newNode.element.line <- l

newNode.element.column <- c

newNode.element.value <- v

node <- newNode

return;

else if node != NIL and v = NIL then

aux <- NULL

if node.element.value = v then

aux <- node.next

@de-allocate node

node <- aux

m.matrix[hash] <- aux

return;

end-if

while node.next != NULL and node.next.element.value != value do

node <- node.next

end-while

if node.next != NULL and node.next.element.value = value

aux <- node.next.next

@de-allocate node.next

node.next <- aux

return;

end-if

else

while node != NULL and node.element.value != value

node <- node.next

end-while

if node != NULL and node.element.value = value do

node.element.value <- v

end-if

end-if

End-Subalgorithm

Complexity:

BC = O(1) – value associated to the key is null

AC = O(

WC = O(1 + *α*) – the value associated to the key is a list of length *α*

Function getElement(m, l, c)

if l > m.noLines or c > m.noColumns or l < 0 or c < 0 then

@Throw InvalidPositionException

end-if

k <- l + c \* m.noColumns

position <- m.hash(k)

node <- m.matrix[position]

if node != NIL then

while node != NIL and node.key != key and node.next != NIL do

node = node.next

end-while

if node == NIL then

telem.line <- -1

telem.column <- -1

telem.value <- 0

return telem

end-if

return node.value

else

telem.line <- -1

telem.column <- -1

telem.value <- 0

return telem

end-if

End-Function

Complexity:

BC = O(1) – the specified element does not exist or it is the head of the list

AC = O(

WC = O(1 + *α*) – the specified element exists on position *α in the list*

Function lineNr(m)

lineNr <- m.noLines

End-Function

Complexity:

BC =AC = WC = ϴ(1) – the number of lines directly accessed at m.noLines

Function columnNr(m)

columnNr <- m.noColumns

End-Function

Complexity:

BC =AC = WC = ϴ(1) – the number of columns directly accessed at m.noColumns

**ADT Tests:**

void Tests::testMine()

{

Mine mine{ 1, 2 };

assert(mine.toString() == "Mine={x=1, y=2, marked=0}");

assert(mine.getX() == 1);

assert(mine.getY() == 2);

assert(mine.isMarked() == false);

mine.setMarked(true);

mine.setX(10);

mine.setY(20);

assert(mine.getX() == 10);

assert(mine.getY() == 20);

assert(mine.isMarked() == true);

}

void Tests::testHashTable()

{

HashTable<int, Mine> table = HashTable<int, Mine>();

Mine mine1{ 1, 2 };

Mine mine2{ 3, 4 };

Mine mine3{ 5, 6 };

Mine mine4{ 9, 10 };

table.insert(1, mine1);

table.insert(24, mine2);

table.insert(2, mine3);

assert(table.getSize() == 3);

table.insert(47, mine4);

assert(table.getSize() == 4);

assert(table.getValue(47) == mine4);

assert(table.getValue(1) == mine1);

assert(table.removeKeyValue(24, mine2) == true);

assert(table.removeKeyValue(24, mine2) == false);

assert(table.removeKeyValue(7, mine2) == false);

assert(table.removeKeyValue(2, mine3) == true);

assert(table.getSize() == 2);

assert(table.update(1, mine1, mine2) == true);

assert(table.update(5, mine1, mine2) == false);

assert(table.update(1, mine4, mine2) == true);

}

void Tests::testSparseMatrix()

{

SparseMatrix matrix = SparseMatrix(5, 5);

assert(matrix.columnNr() == 5);

assert(matrix.lineNr() == 5);

Mine mine;

bool getElemTest = false;

try {

mine = matrix.getElement(6, 6);

} catch (InvalidPositionException ex)

{

getElemTest = true;

}

assert(getElemTest);

mine = matrix.getElement(3, 2);

assert(mine.getX() == -1 && mine.getY() == -1 && mine.isMarked() == false);

mine.setMarked(true);

mine.setX(1);

mine.setY(1);

matrix.modify(1, 1, mine);

assert(matrix.getElement(1, 1) == mine);

Mine newMine{ 2, 4 };

matrix.modify(1, 1, newMine);

assert(matrix.getElement(1, 1) == newMine);

}

**Pseudocode Solution:**

Subalgorithm create(con, size):

con.board <- SparseMatrix(size, size)

con.view <- SparseMatrix(size, size)

con.numberOfMines <- NULL

initMines()

End-Subalgorithm

Complexity:

BC = AC = WC = ϴ (n+m), n = size, m = numberOfMines – time taken to initialize the matrices and the mines

Subalgorithm initMines(con):

con.numberOfMines <- (con.board.columnNr() \* con.board.lineNr()) / 5

emptyElem.x <- -1

emptyElem.y <- -1

emptyElem.value <- 0

for i <- 0, i < con.numberOfMines, 1 do

do

x <- @random % con.board.lineNr()

y <- @random % con.board.columnNo()

while con.board.getElement(x, y) != emptyElem

con.board.modify(x, y, Mine(x, y))

end-for

End-Subalgorithm

Complexity:

BC = AC = WC = ϴ (n), n – numberOfMines – time taken to initialize n mines

Subalgorithm flag(con, x, y):

mine <- con.view.getElement(x, y)

if mine.isMarked() = true then

mine.setMarked(false)

con.view.modify(x, y, mine)

con.board.modify(x, y, mine)

else

mine.setMarked(true)

con.view.modify(x, y, mine)

con.board.modify(x, y, mine)

end-if

End-Subalgorithm

Complexity:

BC =

WC =

AC =

Function expand(con, x, y):

mine.x <- -1

mine.y <- -1

mine.value = 0

if con.board.getElement(x, y).isMarked() then

@throw MarkedSpotException

end-if

if con.board.getElement(x, y) != mine then

expand <- false

else

con.fillBoard(x, y)

expand <- true

End- Function

Complexity:

BC =

AC =

WC =

Subalgorithm fillBoard(con, x, y):

counter <- 0

if y + 1 < con.board.columnNr() and con.board.getElement(x, y+1).getX != -1 then

counter <- counter + 1

if x + 1 < con.board.lineNr() and con.board.getElement(x+1, y).getX != -1 then

counter <- counter + 1

if y + 1 < con.board.columnNr() and x + 1 < con.board.lineNr() and

con.board.getElement(x+1, y+1).getX != -1 then

counter <- counter + 1

if y - 1 > con.board.columnNr() and con.board.getElement(x, y-1).getX != -1 then

counter <- counter + 1

if x - 1 > con.board.lineNr() and con.board.getElement(x-1, y).getX != -1 then

counter <- counter + 1

if y - 1 > con.board.columnNr() and x – 1 > con.board.lineNr() and

con.board.getElement(x-1, y-1).getX != -1 then

counter <- counter + 1

if y + 1 < con.board.columnNr() and x – 1 > con.board.lineNr() and

con.board.getElement(x-1, y+1).getX != -1 then

counter <- counter + 1

if y - 1 > con.board.columnNr() and x + 1 > con.board.lineNr() and

con.board.getElement(x+1, y+1).getX != -1 then

counter <- counter + 1

mine <- Mine(x, y)

mine.setProxy(counter)

con.view.modify(x, y, mine)

End-Subalgorithm

Complexity:

BC =

AC =

WC =

Function getBoardHeight(con):

getBoardHeight <- con.board.lineNr()

End- Function

Complexity:

BC = AC = WC = ϴ (1)

Function getBoardWidth(con):

getBoardWidth <- con.board.columnNr()

End- Function

Complexity:

BC = AC = WC = ϴ (1)