I - Problem statement

This project is about creating a robot control on labyrinth, or specifically finding best suited path for a robot from one point of the labyrinth to another point, using functional programming paradigm and functional programming language such as Haskell.

After solving and finding the best path, an animation should be played showing the robot going through the labyrinth, following a given path. Such animation can be easily done by using one of the many graphical library offered by the Haskell developer community at the hackageDB. In this project, the Gloss graphical library was chosen, thanks to its simplicity for drawing primitive figures such as circle and polygons. At the same time, this project does not require any advanced graphical representation, just a labyrinth (which can be drawn using consecutive squares) and a robot inside (which can be drawn using a smaller square with different color).

This paper is divided into 5 major part:

First part is an introduction to functional programming, the Gloss graphical library and the search algorithm used for solving the problem. It's just an overview of the 4 next parts

Second part goes deeper into the heart of functional programming and the detail about Gloss library. It explain the fundamental of functional programming paradigm, its advantage and drawbacks. And then explain how to draw some primitive figure in Gloss.

Third part explains the algorithm used for searching. These algorithm are not really new, and are mostly just an implementation of algorithm used in imperative programming in a functional way. Data structure used for the programing implementation are discussed in this section too.

Fourth part is about coding and implementation of the algorithm with the use of the Gloss library in order to produce the expected program.

Fifth part is a the testing part, showing the program work, result and performance, followed by the conclusion.

II - Introduction

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It emphasizes the application of functions, in contrast to the imperative programming style, which emphasizes changes in state. Functional programming has its roots in lambda calculus, a formal system developed in the 1930s to investigate computability, the Entscheidungsproblem, function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus.[1]

The language of choice for this programming project is Haskell, as it is a purely functional programming language, easier and the best-known among functional programming languages. Its characteristics will be discussed in later chapter.

Alongside with the Haskell, Gloss was chosen for graphical realization of the project. Gloss hides the pain of drawing simple vector graphics behind a nice data type and a few display functions. Gloss uses OpenGL and GLUT under the hood, and can give a nice result in just a few lines. It do not have many available function, but enough function for this project, as it does not require any advanced graphical manipulation.

But before rendering anything on the screen, a search should be done. It is the heart of the project: finding the best path from one side of the labyrinth to another side. Two search algorithm are used in this project. The first one use recursion while taking advantage of immutability of data in Haskell. The best direction to take from one position is chosen, so that the next position would be the best position. And the next position is recursively chosen, depending on the next best direction, and so on. This algorithm is 2 time slower than the second algorithm used. The second algorithm used the same principle as the breadth first search algorithm. But since Haskell do not allow side effect, then rather than creating a general FIFO type, the algorithm recursively pass the next layer of positions, obtained from the current layer of positions, to the next recursion and continue to pass it down until the position we searched for is among the passed layer. More about the data structure and what I mean by layer will be discussed in the implementation part (part V).

III - Functional and graphical programming with Haskell

A - Functional programming and functional programming languages

1) Definition

Functional programing is a programming paradigm which use mostly execute task by evaluating mathematics mathematical-style expression, rather than a succession of instruction as in imperative programming. Functional programming make an abstraction upon the Neumann computer model, thus dealing with memory state and succession of execution of tasks from memory is avoided in this kind of programming.

A more formal definition from the Haskell official documentation state that, in functional programming, programs are executed by evaluating expressions, in contrast with imperative programming where programs are composed of statements which change global state when executed. Functional programming typically avoids using mutable state.

Functional programming requires that functions are first-class, which means that they are treated like any other values and can be passed as arguments to other functions or be returned as a result of a function. Being first-class also means that it is possible to define and manipulate functions from within other functions. Special attention needs to be given to functions that reference local variables from their scope. If such a function escapes their block after being returned from it, the local variables must be retained in memory, as they might be needed later when the function is called. Often it is difficult to determine statically when those resources can be released, so it is necessary to use automatic memory management.[2] This feature can give a lot of help, as we will see soon in later chapters.

2) Property

- *Higher order function:* it's a function that can either takes one or more functions as an input, or output a function, or take as input then output a function at the same time. This feature is present in Haskell programming language, as in many other different functional programming languages, and is highly used in this project. An example of these functions is `map`, which takes a function and a list as input, and returns a list of the result of the input list, where the input function is applied to them.

- *Purity*: Languages that prohibit side effects are called pure. Side effects happen when a

global variable state are changed by calling a function, or when input variables state change after passing it into a function. In functional programming, such things do not happen, only the return value of the function matters, and that's why Haskell is called purely functional programming language. This purity often implies other properties such as data immutability and lazy evaluation. Data immutability is common in other non-functional language, it is when the internal value of a data cannot be changed, which is an obvious choice for avoiding side effects. Lazy evaluation avoids unnecessary computations and allows, for example, infinite data structures to be defined and used.

- *Recursion:* Recursion is heavily used in functional programming as it is the canonical and often the only way to iterate. Functional language implementations will often include tail call optimization to ensure that heavy recursion does not consume excessive memory.[2] In Haskell, both function and data type can be recursive. Recursive data type and recursive function are already used in many other languages. But in imperative languages, they can be avoided. Instead, in Haskell, recursive function is a must, since there isn't actually any other way to, for example, make a loop.

3) Benefits of functional programming

Functional programming is known to provide better support for structured programming than imperative programming. To make a program structured it is necessary to develop abstractions and split it into components which interface each other with those abstractions. Functional languages aid this by making it easy to create clean and simple abstractions. It is easy, for instance, to abstract out a recurring piece of code by creating a higher-order function, which will make the resulting code more declarative and comprehensible.

Functional programs are often shorter and easier to understand than their imperative counterparts. Since various studies have shown that the average programmer's productivity in terms of lines of code is more or less the same for any programming language, this translates also to higher productivity.[2]

B - Haskell and Gloss

Programming languages such as C/C++/Java/Python are called imperative programming languages because they consist of sequences of actions. The programmer quite explicitly tells the computer how to perform a task, step-by-step. Functional programming languages work differently. Rather than performing actions in a sequence, they evaluate expressions.[3]

Haskell is the best-known pure language for functional programming. Many other impure language are famous, such as Lisp, Scala, Mathematica, oCaml, Curl and even Python has a feature for functional programming. Anyway, Haskell is the best choice for this project, since the goal is to fully use the functional paradigm, and nothing more.

Haskell programs are easier to write, more robust (less buggy), and easier to maintain. The reason is that it use functional programming paradigm and take all of its advantages, such as:

- The absence of side effects, since all data are immutable. The absence of side effects is very helpful when it comes to write a huge program. Side effects make harder to follow the state of all variables, thus easily lead programmers to some bugs.
- Conciseness. An obvious and famous example than can show this is in the quicksort algorithm. Implementation of quicksort algorithm in C++ can take up to 20 lines, while in Haskell it only take 5 lines. Most of the time, programs are 2 to 10 times shorter than its version in imperative languages when written in Haskell. It is an advantage since it makes the code easier to look at once, thus easier to find error.
- High-level. The higher the level of a programming language is, the easier to understand its code will be. The code can be read out almost exactly like the algorithm description. By coding at a higher level of abstraction, leaving the details to the compiler, there is less room for bugs to sneak in.
- Memory managing. As in java, Haskell use garbage collector and programmers don't have to deal manually with pointers anymore. They can worry more about implementing the algorithm.

Apart from all of these enumerated reasons, functional programming itself has other benefits. Functional programming is known to provide better support for structured programming than imperative programming. To make a program structured it is necessary to develop abstractions and split it into components which interface each other with those abstractions. Functional languages aid this by making it easy to create clean and simple abstractions. It is easy, for instance, to abstract out a recurring piece of code by creating a higher-order function, which will make the resulting code more declarative and comprehensible. Functional programs are often shorter and easier to understand than their imperative counterparts. Since various studies have shown that the average programmer's productivity in terms of lines of code is more or less the same for any programming language, this translates also to higher productivity.[2]

Gloss is a 2D-graphical library written in Haskell and use OpenGL and GLUT in order to output graphic. But all of the hardness of OpenGL and GLUT are hidden, leaving a really simplistic API for drawing a simplistic graphics like circle, polygon, loading bitmap pictures and etc.

Even if Gloss does not have many API functions, only one of them will be used and necessary for this project. It is the `animate` function.

`animate :: Display -> Color -> (Float -> Picture) -> IO()` open a new window and display the given animation. Its first parameter is a `Display`, which is for displaying

a window, with given characteristics, or in full screen. The second is a `Color`, it might be `red green blue`, or `rgb`, or `rgba` format. The thirds parameter is a function that can return a `Picture` for each given time. A Picture can be a line, a polygon, a circle, a bitmap; or can be a transformed Picture like a translation, rotation or a scale of another Picture. The time passed to this function is the time since the program started. Therefore, pictures produced by this function will be successively displayed, which will build an animation on the screen. And quite obvious, the return type is a monad IO().

IV - Labyrinth search algorithm

A - Data structure

The goal is to solve a labyrinth case, find path from one point to another. Therefore, it's obvious to take directly a labyrinth as input data. The labyrinth will be stored in file and passed as parameter to the program, so it can find solve the problem according to the labyrinth structure and state.

Labyrinth structure

Labyrinth is just a succession of wall and road. To ease the labyrinth presentation, wall will be presented as 'x' in the input file and road with '.'. At the same time, the labyrinth should contains the information about the starting point and the objective point. The starting point is the position where the robot is initially located. Thus in the labyrinth presentation, in the robot's position will be assigned an 'R', while in the objective position will be assigned a character 'O'.

As a simple example, an input file containing

```
xxxxx
x...x
xRxOx
xxxxx
```

have a visual representation as shown in figure 1.

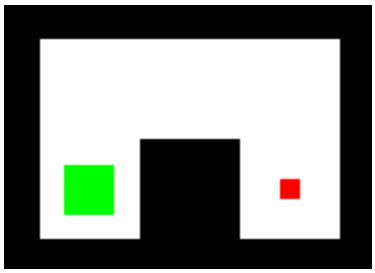

Fig. 1

How this figure is drawn will be discussed in the implementation chapter. The input file will be parsed into list of list of character, or a list of string, in order to make easier the work with the labyrinth and all the information in it. Therefore, from the input file of the above example should give a labyrinth presentation like : `[ 'xxxxx' , 'x...x' , 'xRxOx' , 'xxxxx' ]`

Thereby, to avoid confusion in function definition, it is wiser to define a data type for the labyrinth. Here, it will be called `maze` and defined as follow:

```
type Maze = [String]
```

As we can see, the Maze type is just a synonym for a list of string. Technically, it doesn't make any difference if we use [String] or Maze in definition of function that will work with our labyrinth. But it's a necessary to make easier the understanding of the code.

A position in the labyrinth is given by the position of its respective character in the list of string. So in the above case, the robot is located at (2,1) and the objective at (2,3).

Path and direction

Our goal is to find a path from the location of the robot to some objectives. It is then axiomatic to define this data type. But a path is actually a succession of direction, that is to say a path is just a list of direction. So rather than creating a data type for paths, it's quite logical to create a direction data type.

The robot can go up, down, left or right, according to its position. Seemingly, the data structure for direction and path is defined as follow:

```
data Move = UP | DOWN | LEFT | RIGHT deriving (Show, Eq)
type Path = [Move]
```

Applying move

When a move is applied, the robot position should change, according to the move that was taken. For example:

```
xxxxx             xxxxx             xxxxx
x...x             xR..x             x.R.x
xRxOx ⇒ UP ⇒      x.xOx ⇒ RIGHT ⇒   x.xOx
xxxxx             xxxxx             xxxxx
```

Which is good, but this oversimplification leads to a performance problem. For example, for the labyrinth in the middle, there are two available moves: RIGHT and DOWN. However, going down is not necessary, as we just came from this position. It doesn't actually prevent some algorithm to find the best path (while it can actually make an infinite recursion on a deep-first-search algorithm), nevertheless it will drain the performance down since any algorithm working with such data would explore the DOWN position again, and then would loop UP and DOWN moves. Consequently, we need to encode in the labyrinth presentation that some position have been already explored. For that, we would choose another character to put in these positions, say 'v', after taking a move. So, after a rectification of the above labyrinth change, we should have the following succession:

```
xxxxx             xxxxx             xxxxx
x...x             xR..x             xvR.x
xRxOx ⇒ UP ⇒      xvxOx ⇒ RIGHT ⇒   xvxOx
```

xxxxx xxxxx xxxxx

By doing so, already visited position shouldn't be explored again, since they are not a free road anymore (no '.' character in their places).

Reachability tree

Reachability tree is just a tree, where nodes are labyrinth states, and edges going from one nodes represent available move for the robot from this labyrinth state, and lead to another edges, where the movement were applied.

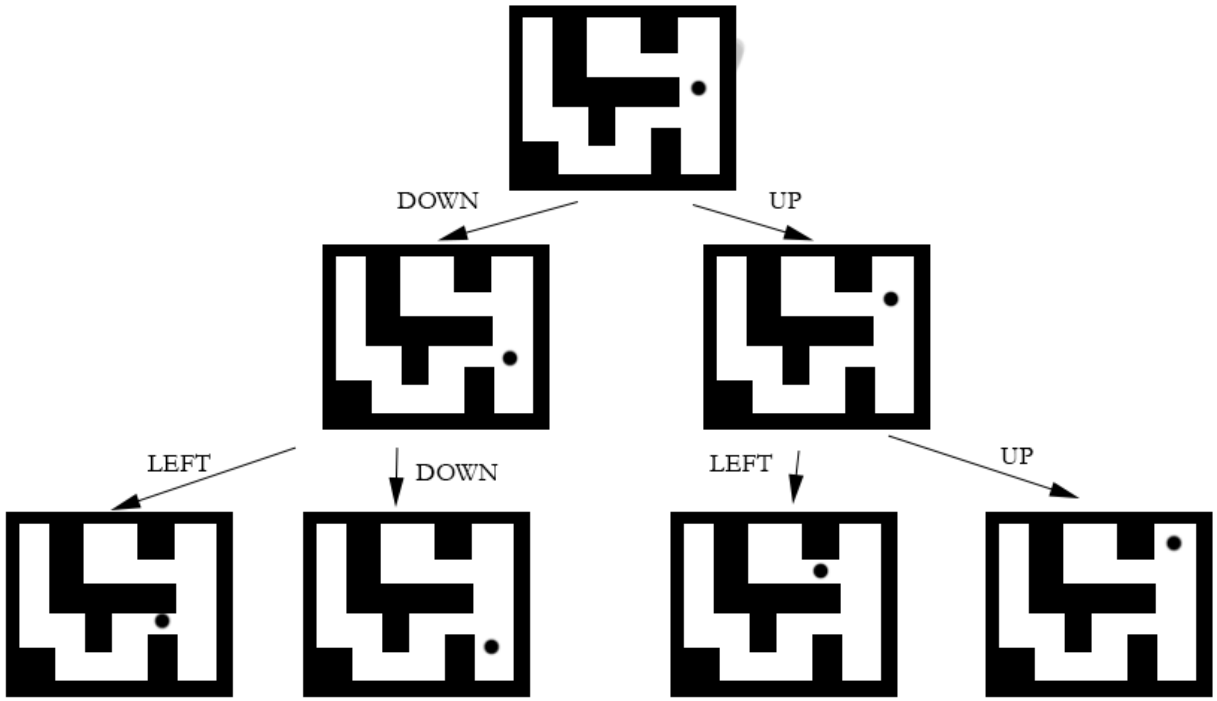

Fig. 2

An example of the reachability tree is shown in the fig. 2. Actually, searching the right path is just searching the right labyrinth state from the root node, and the succession of edges that lead to this labyrinth state.

This search can be done easily using simple recursion, or a more faster algorithm like the implementation of a breadth-first-search.

B - Recursive search

It's hard to say whether this recursive search is the implementation of depth first search or not. But regarding the way it work, it should be, since the recursive search will chose the recursively the move which leads to the next best position, and the next best position is whether the objective position or the position which lead to the next best position. That's how the recursion take place. But actually, one cannot know which of the possible move from the current

position the best is, without going down into the possibility trees. That's why the recursion search works as a depth first search algorithm. However, we cannot confidently say that it is a depth first search, since Haskell use a lazy evaluation. It means that Haskell would evaluate a move only when we need to know whether this move is good or not. Therefore, going down the possibility tree until it finds the best move isn't a necessity in Haskell, thus leaving a depth exploration if not needed anymore.

Shortly, a recursive search is a search in the reachability trees, where the chosen move from a state is the move that leads to the objective state. A state is leads to the objective state if one of his children leads to the objective state, or is the objective state.

C - Breadth-first search

A simplistic definition of the breadth first search algorithm is that breadth-first search algorithm is a search algorithm that begins at the root node and explores all the neighboring nodes.

A more formal definition is:

Breadth-first search (BFS) is a strategy for searching in a graph when search is limited to essentially two operations: (a) visit and inspect a node of a graph; (b) gain access to visit the nodes that neighbor the currently visited node. The BFS begins at a root node and inspects all the neighboring nodes. Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on. [4]

In imperative programming, in order to perform a BFS, one should always use a First In First Out data structure. The search algorithm is very simple:

First, put the root node into the FIFO. Then take a node from the FIFO, until the FIFO is empty or the node that was taken from it is the objective node, if it’s not, then put into the FIFO its successors. If the FIFO is empty and the objective was not found yet, then the search failed.

The reason why it cannot be directly implemented is that FIFO is a mutable object, its content change over time and it is not acceptable in functional programming. On the other side, the FIFO hides the very principle of the BFS, which is the search of an element layer by layer. The BFS should do a search by going progressively further from the root node.

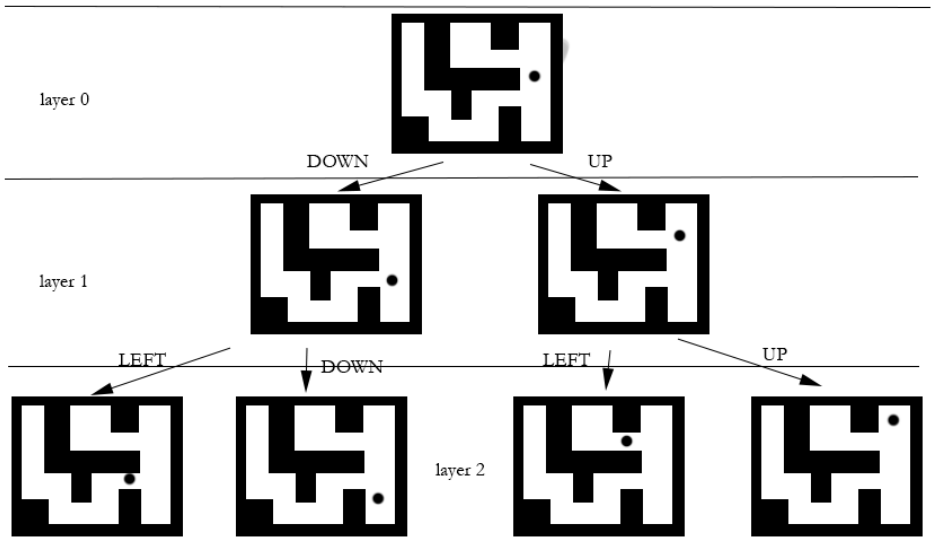

Fig. 3

In Haskell, instead of using FIFO, we would use recursion, which pass each layer of the reachability tree in recursion. The layer is just a list of node that has the same distance from the root note. From each of these layers, it's easy to get the next layer by `map`-ing them to get each node's successors, and then put all of those successors into a list. Each layer would be analyzed if it contains the objective node. If it does, then return the path from the root that can lead to this node, otherwise search into the next layer.

After a benchmark test, it is shown that this algorithm is 2 times faster than the recursive one. And generally, BFS is mostly faster than DFS.

V - Implementation

A - Graphical drawing

After a successful installation of Gloss library, we need only to include 2 modules in our programs: `Graphics.Gloss` and `Graphics.Gloss.Data.Picture.` The first module contains the animate function that we need in order to make animation, and the second contain the data type for creating a Picture.

Drawing primitive figure

A picture constructor from the Gloss hackageDB has the following definition:

```
type Path = [Point]
type Path = [Point]
data Picture =
      Blank
      | Polygon Path
      | Line Path
      | Circle Float
      | ThickCircle Float Float
      | Arc Float Float Float
      | ThickArc Float Float Float Float
      | Text String
      | Bitmap Int Int BitmapData Bool
      | Color Color Picture
      | Translate Float Float Picture
      | Rotate Float Picture
      | Scale Float Float Picture
      | Pictures [Picture]
```

These constructor are quite easy to understand by their name, so we won't go into each specification here.

In order to draw a full maze, one needs to know how to draw a wall and a road. A wall will be just a black square and a road – a white square. Here is the function to for drawing it:

```
drawSquare :: Color -> (Int, Int) -> Picture
drawSquare c (u,v) =
```

```
Translate (caseSizeF*(u+1)) (caseSizeF*(v-1)) $
      Color c (Polygon [(0,0),(0,caseSizeF),(-caseSizeF,
caseSizeF),(-caseSizeF, 0)])
```

In this peace of code, a square would be drawn with a color `c` at coordinates `u` and `v`. Note that `(u,v)` has type `(Int, Int)`. It means that these coordinates are the position of the square to be drawn as described in the labyrinth. `caseSizeF` is the size of a case in pixels. So in order to draw a square with color `c` at the `(u,v)` position, we draw a square of size `caseSizeF` at the coordinate `(0,0)` and then translate it by to the position `(caseSizeF*u, caseSizeF*v)`.

To draw a wall, it is we just need to call `drawSquare black (x,y)`

To draw a road - `drawSquare white (x,y)`

The objective and robot drawing is the same as the wall and road, but with a scaling and different color. A function for drawing an object is then the same as the function for drawing a square but with an extra parameter: the scaling.

```
drawObject :: Color -> (Int, Int) -> Float -> Picture
```

So to draw a robot - `drawObject green (x,y) 0.5`

And to draw an objective - `drawObject red (x,y) 0.2`

Drawing a full maze

Drawing a full maze is just creating a picture from the combination of the picture obtained by drawing each element of the labyrinth.

The function definition is

```
getMazePicture :: Maze -> Int -> Int -> Picture
```

It takes a Maze and its size as input, and output a picture that is corresponding to this maze. So, it draw a road in place of '.' or 'v', a wall in place of 'x', a robot in place of 'R' and an objective in place of 'O'. A full code implementation is available in the annex section.

Opening a window and making an animation

The `animate` function from the `Graphic.Gloss` module takes 3 parameters as already seen above.

```
animate :: Display -> Color -> (Float -> Picture) -> IO()
```

where:

```
data Display = InWindow String (Int, Int) (Int, Int) | FullScreen (Int, Int)
data Color
```

Most of primitive color has a constructor function so we do not have to worry about the rgb distribution. Some of these constructors are `white::Color, black::Color, green::Color, red::Color`

The display define whether the application should be started as fullscreen mode and draw image starting from `(Int, Int)` pixel, or start as window mode with the given name, size and position. In our case we use window mode, and its size is determined by the size of the `Maze`.

The color is the background color.

The third parameter is the function which returns a picture according the time since the program has started. When the path is found by the algorithm, we find the succession of picture that is represented by the path and store it into a list of picture. Then show this picture successively inside the window in every given fraction of seconds, which then produce an animation that show the robot following a path.

B - Search algorithm implementation

Recursive algorithm

```
recursiveSearch :: Maze -> [Move]
recursiveSearch mz = getBestWay mz []
      where
            getBestWay cmz path
                  | isFinal cmz = path
                  | length (getPossibleMove cmz) == 0 = []
                  | otherwise = let getBestWayByMove mv = getBestWay ncmz
(path++[mv]) where ncmz = moveRobot cmz mv
                        in
                              getMinimalListLength $ map getBestWayByMove $
                        getPossibleMove cmz
```

The recursive search function takes as input a maze, and output the best solution.

- `getBestWay :: -> Maze -> [Move] -> [Move]` - takes a maze and a path that leads

to that maze configuration, and returns a path, which is `[]` if there is no solution found for this maze, or returns the solution.

- `isFinal :: Maze -> Bool` – is a function that verify whether the maze is final, meaning that the robot reached all the objective position.

The algorithm implementation is quite straightforward. Haskell chose the best way and append it to the path that lead to the current maze. If the maze is final, then the whole path that led to that maze is return. If there is no other possible move from the robot position of the maze, then the solution was not found and it return an empty list. Otherwise, it should go down the reachability tree, and apply all possible move (`map getBestWayByMove (getPossibleMove cmz)`), and chose the one with the minimal path length if there are more than one solution (`getMinimalListLength :: [[a]] -> [a]`)

Breadth-first search algorithm

Same as the recursive search, this algorithm takes a maze as input, and output the best path found. And what makes this algorithm faster is that the best path found is always the first path found.

```
breadthFirstSearch :: Maze -> [Move]
breadthFirstSearch mz = findIn [(mz,[])]
      where
            findIn mazePathList
                  | length mazePathList == 0 = []
                  | length (succeed mazePathList) > 0 = snd $ head $ succeed
mazePathList
                  | otherwise = findIn $ foldr (++) [] $ map (\(m,p) -> map
(\mv -> (moveRobot m mv, p++[mv])) (getPossibleMove m)) mazePathList
                        where
                              succeed mazePathList = filter (\(maze, path) ->
isFinal maze) mazePathList
```

where:

- `findIn :: [(Maze, [Move])] -> [Move]` – is the main function and find the solution in the current layer given by a list of Maze (or the layer, as seen in the), and the paths that led to these Maze. If the layer is empty, then nothing to search for, so it returns an empty list. If any of the maze in the layer is final, which means that the list returned by succeed mazePathList isn't empty, then returns the path that led to one of the successful maze. Otherwise, create the next layer by applying each possible move of each maze of the layer to that maze, without forgetting to add the taken move into the path that leads to

the next maze (`map (\(m,p) -> map (\mv -> (moveRobot m mv, p++[mv])) (getPossibleMove m)) mazePathList`) . And then, merge all the Maze into a single list before passing it to the next recursion of findIn (`findIn $ foldr (++) [] listOfNextLayers`)

VI – Testing and Analysis

Computer and environment configuration

These algorithms will be tested on a computer with a Dual Core CPU (2.80GHz), with Windows 7 32-bits as operating systems, graphical cards GeForce GT 240 with a total memory of 1233Mb and a display of 1360x768 (32bits).

Input test file

```
xxxxxxxxxxxxx
x.xx...x.0..x
x...0x.x.xx.x
xx.x.x..Rx..x
x..x.x.xxx..x
xx.xxxxx..x.x
x0.x.....0..x
xx...xx.xx.xx
xxxxxxxxxxxxx
```

Screenshots of graphical outputs

Images in the next picture are screenshots taken when the program launched, and then 5s, 10s and 20s after.

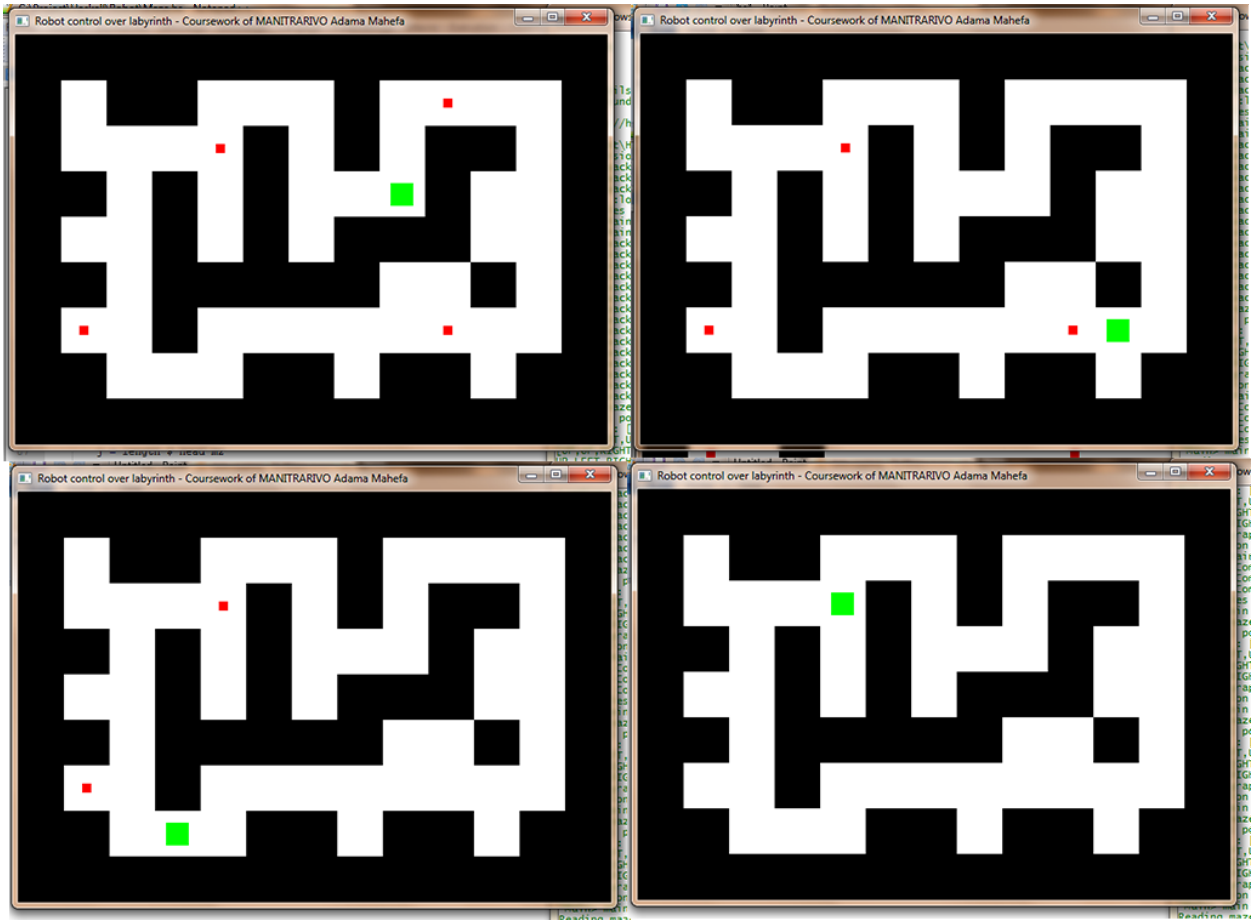

Fig. 4

The program works fine as expected and find the optimal path, which is, in this case:
`[UP, UP, RIGHT, RIGHT, RIGHT, DOWN, DOWN, DOWN, DOWN, DOWN, LEFT, LEFT, LEFT, LEFT, LEFT, LEFT, LEFT, DOWN, LEFT, LEFT, UP, LEFT, RIGHT, UP, UP, UP, UP, RIGHT, RIGHT]`

Both recursive search and BFS returned the same path. However, if we measure the time taken by those algorithms to find the best path, a noticeable difference can be seen.

Number of objective position	RS execution time (ms)	BFS execution time (ms)
1	94	47
2	312	187
3	1186	686
4	2527	1342

VII - Conclusion

Even if functional programming is not mainstream yet, we can see all of its benefits. It can hold a whole algorithm into just a few line of code, and that really make easier the code reading. Any program written in imperative languages can be written in functional programming languages. The main reason why functional languages, such as Haskell, aren't highly used yet is that imperative programming makes it easier to work with computer, which executes tasks imperatively, as all of known operating systems are written using imperative language (which is the C-language). However, a general benchmark over all programming languages show that Haskell program can be a bit faster than its Java fastest version, if implemented well. Plus, programs are robust and easy to maintain in Haskell.

Nonetheless, functional programming is not all about good things; it has its drawback as any languages do. A well-known downside of lazy functional programming, such as Haskell, is that it is very difficult to predict the time and space costs of evaluating a lazy functional program. We encountered such problem while trying to analyze the recursive search function. This problem is fundamental to the paradigm and is not going away. There are excellent tools for discovering time and space behavior post facto, but these tools are mostly reserved for experts.

But functional programming is now on the raise, and starts to catch some companies' attention. Haskell, although initially intended as a research language, has also been applied by a range of companies, in areas such as aerospace systems, hardware design, and web programming.[1]

VIII - Appendices

[1] - http://en.wikipedia.org/wiki/Functional_programming
[2] - http://www.haskell.org/haskellwiki/Functional_programming
[3] - "*Why Haskell Matters*" by Sebastian Sylvan
[4] - http://en.wikipedia.org/wiki/Breadth-first_search

Annex

Main.hs:

```
import Graphics.Gloss
import Maze
import Algo (runAlgo)
import System.Environment (getArgs)
import Control.Monad
import System.CPUTime
import Text.Printf

main =
       do
              start <- getCPUTime
              args <- getArgs
              putStrLn("Reading maze from file xs"++show(args))
              maze <- (getMazeFromFile (head args))
              moves <- (return (runAlgo maze))
              putStrLn(show(moves)++"\nRunning graphics ...")
              end <- getCPUTime
              let diff = (fromIntegral (end - start)) / (10^12)
              printf "Computation time: %0.3f sec\n" (diff :: Float)
              applyMovesOnMaze maze moves
```

Maze.hs:

```
module Maze where

import Debug.Trace
import Data.List
import System.IO
import Graphics.Gloss
import Graphics.Gloss.Data.Picture

type Maze = [String]
data Move = UP | DOWN | LEFT | RIGHT deriving (Show, Eq)
allMoves = [UP, DOWN, LEFT, RIGHT]

caseSize = 50::Int
caseSizeF = 50::Float

applyMovesOnMaze :: Maze -> [Move] -> IO()
drawMaze :: Maze -> IO ()
getCoordinate :: Picture
getMazePicture :: Maze -> Int -> Int -> Picture
```

```
getMazeFromFile :: String -> IO Maze
getMazeSuccession :: Maze -> [Move] -> [Maze]
getNextPosition :: (Int, Int) -> Move -> (Int,Int)
getPosition :: Maze -> Char -> (Int, Int)
getPositions :: Maze -> Char -> [(Int,Int)]
getPossibleMove :: Maze -> [Move]
getRobotPosition :: Maze -> (Int, Int)
moveRobot :: Maze -> Move -> Maze
readMaze :: String -> IO Maze
setPosition :: Maze -> Char -> (Int, Int) -> Maze



readMaze fileName =
       do
              fileHandle <- openFile fileName ReadMode
              contents <- hGetContents fileHandle
              return (lines contents)



getMazePicture m i j =
       let drawRow row y = foldr1 (++) (map (\(c,x) -> drawImage c (x,y)) (zip row
[-fromIntegral(j)/2..]))
              where
                     drawImage t (x,y)
                            | t == '.' = [drawSquare white (x,y)]
                            | t == 'x' = [drawSquare black (x,y)]
                            | t == 'R' = drawImage '.' (x,y) ++ [drawObject green (x,y)
0.5]
                            | t == 'O' = drawImage '.' (x,y) ++ [drawObject red (x,y)
0.2]
                            | otherwise = [drawSquare white (x,y)]
                            where
                                   drawSquare c (u,v) = -- trace ("drawing "++show(c)++"
square at ("++show(u)++","++show(v)++")") $
                                                 Translate (caseSizeF*(u+1)) (caseSizeF*(v-1)) $
Color c (Polygon [(0,0),(0,caseSizeF),(-caseSizeF, caseSizeF),(-caseSizeF, 0)])
                                   drawObject c (u,v) sc = Translate (caseSizeF*(u+tmv))
(caseSizeF*(v-tmv)) $ Scale sc sc $ drawSquare c (0,0) where tmv = (1-sc)/2
       in
              Pictures $ foldr1 (++) $ map (\(r,n) -> drawRow r n) (zip m (iterate (-1+)
(fromIntegral(i)/2)))



getCoordinate =
       let m = 1000
       in Pictures $ [Line [(0,-m),(0,m)], Line[(-m,0),(m,0)]] ++ (positionOverY m) ++
(positionOverX m)
       where
              positionOverX m = map (\t -> Translate (100*t) 0 (Scale 0.1 0.1 (Text (show
```

```
(100*t))))) [-m/100 .. m/100]
              positionOverY m = map (\t -> Translate 0 (100*t) (Scale 0.1 0.1 (Text (show
(100*t))))) [-m/100 .. m/100]



drawMaze m = display (InWindow "My Window" (j*caseSize, i*caseSize) (10, 10)) (greyN 0.5)
$
       getMazePicture m i j
       where
              j = length $ head m
              i = length m



getMazeFromFile f = ((openFile f ReadMode) >>= \fh -> hGetContents fh) >>= \contents ->
return (lines contents)

frameSec = 0.5::Float
applyMovesOnMaze mz mvs = --trace ("Applying moves : "++show(mvs))
       animate (InWindow "Robot control over labyrinth - Coursework of MANITRARIVO Adama
Mahefa" (j*caseSize, i*caseSize) (10, 10)) (greyN 0.5) $
       (\time -> getMazePicture ((getMazeSuccession mz mvs) !! (frameNum time)) i j )
       where
              k time = if floor(time/frameSec) < 0 then 0 else floor(time/frameSec)
              j = length $ head mz
              i = length mz
              frameNum time = min (length mvs) (k time)



getMazeSuccession mz mvs =
       let ccat (mzlist, maze) move = (mzlist ++ [nextMaze], nextMaze)
              where nextMaze = moveRobot maze move
       in
              fst $ foldl ccat ([mz], mz) mvs



moveRobot mz mv = --trace ("moving robot from position "++show(curX, curY)++"
"++show(mv)) $
       setPosition (if mz !! nextX !! nextY == 'O' then clearMaze mz else setPosition mz
'v' (curX, curY)) 'R' (nextX, nextY)
       where
              (nextX, nextY) = getNextPosition (curX, curY) mv
              (curX, curY) = getRobotPosition mz
              clearMaze m = map (\s -> map (\c -> if c=='v' || c=='R' then '.' else c) s)
m



getNextPosition (x, y) m
       | m == UP = (x-1, y)
```

```
        | m == DOWN = (x+1, y)
        | m == LEFT = (x, y-1)
        | m == RIGHT = (x, y+1)

getPositions mz c = foldr (++) [] $ map (\l ->  zip [l,l..] (getContainingColumns mz l))
$ getContainingLines mz
        where
                getContainingLines m = filter (>=0) (map (\(a,b) -> if c `elem` a then b
else -1) (zip m [0..]))
                getContainingColumns m l = filter (>=0) (map  (\(a,b) -> if a==c then b
else -1) (zip (m !! l) [0..]))

getPosition mz c = head $ getPositions mz c

getRobotPosition mz = getPosition mz 'R'

setPosition mz w (x,y) = map (\(a,b) -> if b==x then map (\(c,d) -> if d==y then w else
c) (zip a [0..]) else a) (zip mz [0..])

getPossibleMove mz =
        filter (\mv -> notWall mz (getNextPosition (x,y) mv)) allMoves
        where
                notWall mz (x,y) = (mz !! x !! y == '.') || (mz !! x !! y == 'O')
                (x,y) = getRobotPosition mz
```

Algo.hs:

```
module Algo where

import Debug.Trace
import Maze

runAlgo :: Maze -> [Move]
recursiveSearch :: Maze -> [Move]
breadthFirstSearch :: Maze -> [Move]

isFinal :: Maze -> Bool
getMinimalListLength :: [[a]] -> [a]

runAlgo m = recursiveSearch m

recursiveSearch mz = getBestWay mz []
        where
                getBestWay cmz path
                        | isFinal cmz = path
                        | length (getPossibleMove cmz) == 0 = []
                        | otherwise = let getBestWayByMove mv = getBestWay ncmz (path++[mv]) where
ncmz = moveRobot cmz mv
                        in getMinimalListLength $ map getBestWayByMove $ getPossibleMove cmz
```

```
breadthFirstSearch mz = findIn [(mz,[]::[Move])]
        where
                findIn mazePathList
                        | length mazePathList == 0 = []
                        | length (succeed mazePathList) > 0 = {-trace ("DONE WITH"++show(snd $ head $
succeed mazePathList))-} snd $ head $ succeed mazePathList
                        | otherwise = findIn $ foldr (++) [] $ map (\(m,p) -> map (\mv -> (moveRobot m mv,
p++[mv])) (getPossibleMove m)) mazePathList
                                where
                                        succeed mazePathList = filter (\(maze, path) -> isFinal maze)
mazePathList


getNextConfiguration (mz, (x, y)) mv = (moveRobot mz mv, getNextPosition (x,y) mv)

getMinimalListLength t =
        let
                sortByLength [] = []
                sortByLength (a:l) = (sortByLength $ filter (\ll -> length ll < length a) l) ++ [a] ++
(sortByLength $ filter (\ll -> length ll > length a) l)
                filteredList l = filter (\ll -> 0 /= length ll)  l
        in
                if length (filteredList t) == 0 then []  else head $ sortByLength (filteredList t)

isFinal mz = length (getPositions mz 'O') == 0
```