Universiteit van Amsterdam

Data Structures

Maze Solver

Author:
Abe Wiersma

February 23, 2013

Contents

1 Preface

For the course data structures we got the assignment to make a maze solver implementing at least 2 algorithims this is the documentation on this assignment. Skeleton code was also given.

2 Problems

To make the maze solver there are a few problems to be solved, this section shall contain all the problems that needed solving.

2.1 Reading the maze file

For this assignment we got maze files in this format:

```
14,14
##############
#   ####     #
# #     S#####
# ########   #
#    #E    # #
# ########## #
# #   #      #
# ### # #### #
#  ## #    ###
## ## #### ###
## ##        #
##    ###### #
## ####      #
##############
```

So reading files like this is the first problem.

2.2 Walking through the maze

To solve a maze like the one provided above you also need a walker to move through the file ending when the exit is found.

2.2.1 Algorithims to give directions to walker

For this assignment you had to implement two simple maze solving algorithms:

- A random walker
- A wall following walker

2.2.2 Printing the steps

Part of the assignment is to also return with which moves the walker does.

3 Solutions

And now my solutions

3.1 Reading the maze file

For the reading of the maze file the function readMaze in maze.c is used, the function returns a struct containing a char array containing the maze, the start position, the end position and the width and height of the maze.

First the full file is read into a temporary char array, then the width and height are read from the top of the char array. The width and the height are given to the initMaze function which returns a initialized struct with the char array malloc'ed to be able to contain the maze. After the struct is initialized the char array is filled char by char. When an 'S' or 'E' is found the position is registered in the struct.

3.2 Walking through the maze

The walker is a struct containing a x position and a y position. A walker is initialized at the startposition retrieved from the maze struct. To move the walker there is a function called moveWalker which requires a direction the maze struct and the walker struct. If a step is valid the walker struct's positions are modified to the new position and a 1 is returned to say the move was made. If the move could not be done a 0 is returned.

3.2.1 Algorithims to give directions to walker

So as discussed before i had to implement at least the:

- Random algorithm
- Wall following algorithm

But i also added my own algorithm which i shall explain further on.

The random walker generates a random number between 0 to 3 and keeps doing this until a valid move is found. The function uses the rand() function from the stdlib.

```
while(!moveWalker(rand() % 4, walker, maze));
```

The wall following algorithm uses the last direction and rotates to the left if there's a wall there rotate to the right until there's a blank.

```
while (!moveWalker(direction, walker, maze))
{
    direction = (direction + 1) % 4;
}
```

The algorithm i added myself i will be explaining with visual help.

```
########
#E     #
###S####
##   ###
########
```

This will be our explanation maze.

First the algorithm starts of by copying the size of the char array to a int array. The int Array will look like this.

```
-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1
```

Then the start position gets defined in the int Array, this is step 0.

```
-1-1-1-1-1-1-1-1    ########
-1-1-1-1-1-1-1-1    #E     #
-1-1-1 0-1-1-1-1    ###0####
-1-1-1-1-1-1-1-1    ##   ###
-1-1-1-1-1-1-1-1    ########
```

Now the looping begins until the end is found. The int array changes like these steps...

```
-1-1-1-1-1-1-1-1    ########
-1-1-1 1-1-1-1-1    #E 1   #
-1-1-1 0-1-1-1-1    ###0####
-1-1-1 1-1-1-1-1    ## 1 ###
-1-1-1-1-1-1-1-1    ########

-1-1-1-1-1-1-1-1    ########
-1-1 2 1 2-1-1-1    #E212  #
-1-1-1 0-1-1-1-1    ###0####
-1-1 2 1 2-1-1-1    ##212###
-1-1-1-1-1-1-1-1    ########

-1-1-1-1-1-1-1-1    ########
-1 3 2 1 2 3-1-1    #32123 #
-1-1-1 0-1-1-1-1    ###0####
-1 3 2 1 2-1-1-1    ##212###
-1-1-1-1-1-1-1-1    ########
```

At this point the end is found and the looping stops. You can see the int array respecting the walls in the char array. Now an int array given to the algorithm is malloc'ed with the number located in the endposition(this is also the number of steps required). All the directions are stored in this int array. It is filled starting from the endpoint, looking for the previous number.
I will now demonstrate with the example how the int array is filled:

Int array when initialized:

```
[] [] []
```

Beginning from the end:
To the West.

```
[] [] [3]
```

To the West again.

```
[] [3] [3]
```

To the North.

```
[0] [3] [3]
```

The number of steps is returned if an exit is found, if no exit is found a 0 is returned.

The int array that was previously given to the algorithm is walked through, and the shortest route is found.

3.2.2 Printing the steps

Every step the maze with the position of the walker is printed by the function printMaze, this is done by first copying the char map from the maze struct into a temporary char array in which the position of the walker retrieved from the walker struct is changed to an 'X'. Every step a maze is printed to console.