UNIVERSITEIT VAN AMSTERDAM

DATA STRUCTURES

Maze Solver

breadth first

Author:
Abe WIERSMA

March 22, 2013

Contents

1 Preface

For the course data structures we got the assignment to make a maze solver which finds the shortest path, this is my execution of the assignment. For this assignment we got some skeleton code for reading a maze, and code to create a spanning tree.

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.

2.1.1 Finding the shortest path

Well this one speaks for itself really, but there are several different ways to do this.

3 Solutions

And now my solutions .

3.1 Reading the maze file

Because this time the skeleton code provided had a fully operational maze reader which reads a maze into a struct:
- The maze itself is stored in a two dimensional char array called map.
- The dimensions of the array are stored in the int's nrows and ncols.
Because I wanted the struct to store the position of the exit and the start i

added a function in the maze.c and I added four int's which store de positions of the start and exit.

3.1.1 Finding the shortest path

The algorithm I used to find the shortest path is the breadth first algorithm. I made this algorithm for assignment 3 which also involved solving a maze. I shall be explaining it with visual help.

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

This will be our explanation maze.

The algorithm starts of by making a 2-d int-array with the same size of the char array in the struct. All values in this 2-d array are set to -1.

```
-1-1-1-1-1-1-1-1 ########
-1-1-1-1-1-1-1-1 #E     #
-1-1-1-1-1-1-1-1 ###S####
-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.

So at this point there are numbers 0 to 3 making a path to the exit. Because the location of the exit is stored we can easily walk back number for number from this position. This is done like so.

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

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

In a large maze this would look like this.... (this example is the solution of map3.txt given to us with the assignment.)

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

```
# ## # #    ##   ##    # # #    # ## #
# #  # ############# # ### ### ##    #
### ##      #     ####  #   #   #### #
#   ## ## #   # #  ##  ## ### #    # #
# ###   ##########  # ##  # # #### # #
#   #####   ##   ##   ## ## #   #### #
# #    ## #  # # ######     ###    # #
# # ##    ##   #        # #   # ##   #
#####################################
```

4 Conclusion

In the skeleton code we were provided with a way to implement a tree which I started of with, but after a while I thought the simplicity of my previous algorithm had its charm, and I recoded it to work with the new maze struct. I was pretty far advanced with the implementing of the tree: I had thought of a way to implement an array of nodes similair to the int array, but not to fill the walls with a value as I did with the int array. Then to follow the path back from the exit node to the start node parent after parent.

In the archive i will also put two generated mazes given to me by my good friend Pjotr Buys whom made a generator, this to show it also works with mazes of a size 1000x1000, and just a proof of concept.