COMP 2160 – Assignment 2

Programming practices

Due March 5th, 2020 at 11:59pm

Description

In this assignment you will write a program that implements a *backtracking algorithm*. Implementing this program will involve the use of simple data structures (a linked list) and dynamic memory allocation. You will be expected to apply the principles of Design by Contract to your code.

Notes

- Read the *entire* assignment document before starting.
- Your program must run correctly both with assertions enabled and disabled (when compiled with and without the `-DNDEBUG` option).
- Please be sure to follow the programming standards; not doing so will result in a loss of marks, you will find the programming standards in UMLearn.
- If you did not accept the Honesty Declaration Checklist, you need accept it to get grades.
- All submitted assignments will be scanned with automated software to monitor for academic integrity violations.
- Your assignment code must be handed in electronically. See the Submitting your assignment section below for more information.
- Late assignments are not accepted. The due date is enforced electronically.

Objectives

- Dynamic memory allocation and resource management.
- Applying the principles of Design by Contract.
- Implementing data structures idiomatically in C (with addresses and pointers)

Question 1: 🐭 Maze

Your goal is to write a program that will test, using a *backtracking algorithm*, if a mouse can escape from a rectangular maze. To ensure consistency of design, start your solution with `maze_start.c`.

General idea

The backtracking algorithm helps the mouse by systematically trying all the routes through the maze until it either finds the escape hatch or exhausts all possible routes (and concludes that the mouse is trapped in the maze). If the backtracking algorithm finds a dead end, it retraces its path until it reaches a position from which there is an untried path. The backtracking algorithm always tries all directions from any position, and always in the same order.

Input

The input to the algorithm is a maze with walls (represented by `'1'` characters) and open passage ways (represented by `'0'` characters). The starting position of the mouse is represented by `'r'` and the escape hatch from the maze by `'e'`. The first line of the input will contain the number of rows and the number of columns in the maze. Here's an example of what a maze might look like as input:

```
6 5
1 1 1 1 1
1 0 0 e 1
1 1 1 0 1
1 r 1 0 1
1 0 0 0 1
1 1 1 1 1
```

The maze will **always** have a wall around the outside, so you need not be concerned about the mouse falling off the maze as it explores all directions.

Store the maze in a two-dimensional array but be careful that your program avoids subscripting your array "out of bounds". Not checking for array boundaries may lead to your program crashing, or worse, giving the wrong results. A document is uploaded to the assignment's folder in UMLearn detailing what you need to know about using two-dimensional arrays.

To develop your program, you can use the input file `testMaze.txt`, which contains the above maze. There are also other input files provided in UMLearn so that you can test your solution for different inputs. All input for this program should be done on standard input (**NO** `fopen` is required for this problem).

The algorithm

The backtracking algorithm keeps a list of positions that are the beginnings of paths it has yet to try. From the current position, the algorithm adds any untried open neighbouring positions (if there are any), always looking forward, backward, left and right from the current position. At each step, the algorithm gets the next position from the list and adds into the list the untried neighbouring positions. Finally, the algorithm must mark each visited position with a period (`'.'`) to avoid revisiting positions – so that it will not loop forever trying the same routes.

Here's a pseudocode description of the algorithm:

```
read in the maze;
initialize the list;
goalCell = the position of the escape hatch in the maze;
startCell = the initial position of the mouse in the maze;
currentCell = startCell;
while currentCell is not the goalCell
  mark currentCell as visited;
  add to the list the unvisited open neighbours of currentCell;
  if the list is empty
    the mouse is trapped: we tried all routes and failed to find an escape;
  else
    get the next cell from the list and make it currentCell;
end while;
the mouse can escape the maze: we reached the goal cell.
```

To support the backtracking algorithm, you will need to implement a linked list. To work correctly, all you need is to always add a cell at the top of the list and always get the next cell from the top of the list (no traversals or ordering required – what we are building is called a *stack*).

Output

Your program must print out the maze after each cell is visited, showing which cells have already been visited. Finally, your program will print out a message indicating whether an escape was found or that the mouse is trapped. Sample output can be found in the file `testMazeOutput.txt`.

All output for this program should be done on standard output (**NO** `fopen` or `fprintf` required for this problem).

Submitting your assignment

Create a directory called `comp2160-a2-<yourlastname>-<yourstudentid>` and place your `.c` files inside the directory.

Then run the command:

```
handin 2160 a2 comp2160-a2-<yourlastname>-<yourstudentid>
```

- You may *optionally* include a `README` file in your directory that explains anything unusual about compiling or running your programs.
- You may resubmit your assignment as many times as you want, but **only the latest** submission will be kept.
- We only accept homework submissions via `handin`. Please **do not** e-mail your homework to the instructor or TAs -- it will be ignored.

- We reserve the right to refuse to grade the homework, or to deduct marks if these instructions are not followed.

Evaluation

Your assignment is worth 10 points, 5 points × 2 for question 1.

Note that your code **must** compile -- no part marks will be given for code that does not compile.

Warnings

The code you write should compile with `clang -Wall` *without* any warnings. A 1-point penalty will be applied to your score (before multiplication) for *each* warning that is emitted by the compiler.

Question 1

Level	Criteria

0	• No submission for Q1 is made, or
	• The submitted code does not compile on `rodents.cs.umanitoba.ca` with the expected compilation commands.
	
	`clang -Wall <yourfile.c> -o <yourfile>`
	`clang -Wall -DNDEBUG <yourfile.c> -o <yourfile>`

1	• Code is submitted for the problem, and the code compiles. • The compiled code crashes, does not run, or does not complete in a reasonable amount of time (< 5 seconds) when executed with a file being supplied with the redirection operator (<). • The C code is of high quality, following the standards described in the programming standards document.

Level	Criteria
2	• The criteria for 1 is met. • The program works with the redirection operator (<). • The `loadMaze` function is implemented correctly. • DbC principles are applied to `loadMaze` (pre-conditions, invariant on state of maze applied as post-condition).
3	• The criteria for 2 is met. • The `printMaze` function is implemented correctly. • DbC principles are applied to the `printMaze` function (pre- and post-conditions).
4	• The criteria for 3 is met. • The data structure for the maze solving code is *complete*, along with supporting functions (`makeCell`, `addCell`, `nextCell`) • DbC principles are applied to the maze solving data structure (an invariant) and supporting functions (pre- and post-conditions).
5	• The criteria for 4 is met. • The maze solving code is complete and works correctly. • DbC principles are applied to the maze solving code (pre- and post-conditions).