**1. Questions (include these in your write-up):**

1a. Meets specifications aside from error output descriptions being a little different.

1b. (.5 point) Copy and paste your terminal window contents from running solve on the 15 mazes

(similar to what is copied/pasted above for sample output).

[CAMPUS\liao18@egr1 hm2]$ ./solve maze1.txt

success in storing maze!

Maze can be solved.

[CAMPUS\liao18@egr1 hm2]$ ./solve maze2.txt

success in storing maze!

Maze cannot be solved.

[CAMPUS\liao18@egr1 hm2]$ ./solve maze3.txt

success in storing maze!

Maze cannot be solved.

[CAMPUS\liao18@egr1 hm2]$ ./solve maze4.txt

success in storing maze!

Maze can be solved.

[CAMPUS\liao18@egr1 hm2]$ ./solve maze5.txt

Error reading maze. Given num\_cols is not equal to actual value. Cannot create maze

[CAMPUS\liao18@egr1 hm2]$ ./solve maze6.txt

Invalid row size. Cannot create maze

[CAMPUS\liao18@egr1 hm2]$ ./solve maze7.txt

Error reading maze. Given num\_cols is not equal to actual value. Cannot create maze

[CAMPUS\liao18@egr1 hm2]$ ./solve maze8.txt

Actual number of rows doesn't equal given maze height. Cannot create maze

[CAMPUS\liao18@egr1 hm2]$ ./solve maze9.txt

success in storing maze!

Maze cannot be solved.

[CAMPUS\liao18@egr1 hm2]$ ./solve maze10.txt

Invalid column start location. Cannot create maze

2c. (1 point) What types of mazes did you create for maze11, maze12, and maze13? Ones that have solutions? Ones with improper formatting? Ones with more/fewer than 6 ints on the top row?   
In your answer for part (c), make a table for the new mazes:

Table 1: Testing of new mazes

|  |  |  |
| --- | --- | --- |
| Filename | What is wrong (or correct) about the file? | What your program did when running on this file |
| maze11.txt | Maze11 invalid col given. | Saw that actual number of columns doesn’t equal given number |
| maze12.txt | Maze12 invalid rows given. | Saw that actual number of rows doesn’t equal given number |
| maze13.txt | Maze13 invalid start location given. | Saw that actual start location doesn’t equal given coordinates. Maze couldn’t be solved |
| maze14.txt | Maze14 invalid exit location given. | Saw that actual exit location doesn’t equal given coordinates. Maze couldn’t be solved. |
| maze15.txt | Maze15 cannot be solved. | There is no exit in the maze. |

2d. (.5 point) Do you have test cases for all possible improperly formatted mazes? If not, state the cases that you did not test.

I did not test that the top row doesn’t check for all six numbers.

3. (2 points) Suppose the following small maze is used for the solve function:

\*\***\***\*\*

\* **\***\*\*

\*\* \*\*

with row = 0 and col = 2 to start and exit point (2, 2).

What recursive calls are made when solve executes? Complete the tree. If a recursive call ends in false, you may quit that branch. You can save writing by using s(0,2,m) to stand for solve(0,2,maze). Remember that the code puts ‘\*’ in maze positions that are already examined. So, when solve(0, 2, maze) is executed, the char at row 0 col 2 is assigned ‘\*’.

solve(0, 2, maze)

solve(0, 1, maze) solve(-1, 2, maze) solve(0, 3, maze) solve(1, 2, maze)

wall (return 0) off maze (return 0) wall (return 0)

solve(0,2,maze) solve(2,2,maze) solve(1,1,maze) solve(1,3,maze)

wall(return 0) wall(return0)

exit(return 1) solve(0,1,maze) solve(2,1,maze) solve(1,0,maze) solve(1,2,maze)

wall (return 0)

3a. (.25 pt) How much time did you spend in total on this homework assignment (including the report)?

About 7 hours of working including the report.

3b. (.25 pt) What was the most challenging part for you when completing this assignment?

Debugging. It is challenging to identify problems when the recursive function is broken.

**Appendix A:** I verify that the code and this write-up were authored by me. I have documented the help I have received in comments in the code files.

**Appendix B**: Copy and paste your maze.c file and makefile here (use Courier New 8pt font so the characters line up correctly)

Main.c

/\* Author: Tammy VanDeGrift

CS 305 Spring 2016

Homework 2: mazes

This is the main function: it opens the file specified at the command line,

creates a maze object, and determines if the maze is solvable.

\*/

#include <stdio.h>

#include <stdlib.h>

#include "maze.h"

#define NUM\_PARAMS 2

/\* prototype \*/

void usage(char \* executable);

/\* main

executable\_name input\_filename.txt

opens input\_filename.txt for reading

creates maze object

runs the maze solver

frees maze

\*/

// function completed for the CS 305 students

int main(int argc, char \* argv[]) {

if(argc != NUM\_PARAMS) {

usage(argv[0]);

return EXIT\_FAILURE;

}

// open file for reading

FILE \*fp = NULL;

fp = fopen(argv[1], "r");

if(fp == NULL) {

fprintf(stderr, "Error opening input file %s. Exiting.\n", argv[1]);

return EXIT\_FAILURE;

}

// create new maze object

maze \*the\_maze = create\_maze(fp);

// check maze to see if it was created successfully

if(the\_maze == NULL) {

return EXIT\_FAILURE;

}

// done with file at this point

fclose(fp);

fp = NULL;

// run the solve function from the maze entry position

int can\_solve = solve(the\_maze->entry.row, the\_maze->entry.col, the\_maze);

if(can\_solve == 0) {

printf("Maze cannot be solved.\n");

} else {

printf("Maze can be solved.\n");

}

// free memory and exit

free(the\_maze);

return EXIT\_SUCCESS;

}

/\* usage

prints error message to user

\*/

void usage(char \* executable) {

printf("Usage: \n%s maze\_file.txt\n", executable);

}

MAZE.C

#include <stdio.h>

#include <stdlib.h>

#include "maze.h"

maze \* create\_maze(FILE \*in) {

maze \* mazeIn = malloc( sizeof(maze));

//start line

char \* line = malloc(sizeof(char)\*100);

// start line string size at 100

int num\_rows;

int num\_cols;

int start\_row;

int start\_col;

int end\_row;

int end\_col;

ssize\_t num\_read;

num\_read = getline(&line, &num\_read, in);

if(sscanf(line, "%d %d %d %d %d %d", &num\_rows, &num\_cols, &start\_row, &start\_col, &end\_row, &end\_col) != 6) {

fprintf(stderr, "Maze file format incorrect. Top line must include 6 numbers.\n");

return NULL;

}

//check for invalid maze parameters

if (num\_rows <= 0) {

fprintf(stderr, "Invalid row size. Cannot create maze\n");

return NULL;

}

if (num\_cols <= 0) {

fprintf(stderr, "Invlaid column size. Cannot create maze\n");

return NULL;

}

if (start\_row < 0 || start\_row >= num\_rows) {

fprintf(stderr, "Invalid row start location. Cannot create maze\n");

return NULL;

}

if (start\_col < 0 || start\_col >= num\_cols) {

fprintf(stderr, "Invalid column start location. Cannot create maze\n");

return NULL;

}

if (end\_row < 0 || end\_row > num\_rows) {

fprintf(stderr, "Invalid row exit location. Cannot create maze\n");

return NULL;

}

if (end\_col < 0 || end\_col > num\_cols) {

fprintf(stderr, "Invalid col exit location. Cannot create maze\n");

return NULL;

}

//valid tests passed. Assign maze values

mazeIn->width = num\_cols;

mazeIn->height = num\_rows;

mazeIn->entry.row = start\_row;

mazeIn->entry.col = start\_col;

mazeIn->exit.row = end\_row;

mazeIn->exit.col = end\_col;

//create heap of "data" for mazeIn, 2D array

int i;

int j;

mazeIn->data = (char \*\*)calloc(num\_rows, sizeof(char \*));

if(mazeIn->data != NULL) {

/\* allocate memory to store data for each row \*/

for(i = 0; i < num\_rows; i++) {

mazeIn->data[i] = (char \*)calloc(num\_cols, sizeof(char));

}

}

//get each line. Error check the following before storing in data

// 1) That actual rows count matches given maze height

// 2) That actual columns count match given maze width

// 3) That there are no illegal characters in the maze: non ' ' or '\*'

int c; //for EOF value storage

for(j = 0; j < num\_rows ; j++) {

num\_read = getline(&line, &num\_read, in); //read new line

if(num\_read != (num\_cols +1) ) { //check that this line is valid length (valid # of cols)

fprintf(stderr, "Error reading maze. Given num\_cols is not equal to actual value. Cannot create maze\n");

return NULL;

}

for(i = 0; i < num\_cols; i++) { //put line inside ith line of mazeIn.data

if(line[i] != ' ' && line[i] != '\*') { //check for illegal character in maze

fprintf(stderr, "Illegal character that is not space or \*. Cannot create maze\n");

return NULL;

}

//debugging code here to print out stored maze in case something is broken

mazeIn->data[j][i] = line[i];

//printf("%c",mazeIn->data[j][i]);

}

//printf("\n");

}

if( (c = fgetc(in) ) != EOF) {

fprintf(stderr, "Actual number of rows doesn't equal given maze height. Cannot create maze\n");

return NULL;

}

printf("success in storing maze!\n\n");

return mazeIn;

}

int solve(int row, int col, maze \* the\_maze) { //recursive function of 4 cases.

if( (row < 0) || (row >= the\_maze->height) || (col < 0) || (col >= the\_maze->width)) {//going out of bounds. No solution

//printf("case 1\n");

return 0;

}

else if( (row == the\_maze->exit.row) && (col == the\_maze->exit.col) && (the\_maze->data[row][col] == ' ') ) {//found the exit

//printf("case 2\n");

return 1;

}

else if( (the\_maze->data[row][col] == '\*') ) {//ran into a wall

//printf("case 3\n");

return 0;

}

else {

//printf("case 4\n"); //if there is an empty space, then we can go there! Use recursive function to find possible solutions from there

the\_maze->data[row][col] = '\*';

return (solve(row, col-1, the\_maze)) || (solve(row-1, col, the\_maze)) || (solve(row, col+1, the\_maze)) || (solve(row+1, col, the\_maze));

}

}

void free\_maze(maze \* the\_maze) {

//create heap of "data" for mazeIn, 2D array

int i;

for(i = 0; i < the\_maze->height; i++) {

free(the\_maze->data[i]);

}

free(the\_maze->data);

free(the\_maze);

}

MAKEFILE

solve: main.o maze.o

gcc -o solve main.o maze.o

%.o: %.c %.h

gcc -c $<

clean:

/bin/rm -f \*.o solve