Data Structure Assignment#2

German language and literature 2016130927

Park Jun Yeong

1. Development environment: Microsoft Visual Studio 2013

2. Explanation and algorithm for the code

Since how the maze is structured is unknown in advance, it should allocate the size of maze dynamically and copy the input text file, which was passed as a first command-line argument when this program is executed by user. For the convenient path-finding, you should another 2 rows and 2 columns so that the original maze should be blocked on all four sides by 1. In addition to this maze, another two dimensional array Mark, exactly same sized with Maze, should be initialized as 0 in advance in order to check whether a certain position is visited or not.

After having this maze model, the path from starting position to exit position is to be searched, checking recursively whether next position is blocked (if it is same with 1?) and not visited. In this C code, how the next position is determined is according to checking 8 directions from North to Northwest clockwise. This recursive path-finding based on DFS algorithm pushes its next position’s coordinate value to stack when its next position becomes the exit position-only the moment it successfully passes the maze. Since then, it pushes its indexes to stack from ‘exit position’ to ‘starting position’ (to be exact, to very next position of the starting point at this moment. Only after these are pushed, the coordinate value of starting point would be pushed. So whether its path is found or not, starting position is always pushed at the last time.): So currently top of stack is indicating the starting position and its last node has exit position (if the path is found). Plus, this stack would be also allocated dynamically as needed: it is implemented by linked list.

At this moment, the path is given: in the output text file, which was passed as a second command-line argument, these should be written. Since the indexes from starting to exit are to be written sequentially, they just need to pop in the order they are stored in the stack: remember that the top of stack indicates the starting position and last node has an exit position if this maze is solvable. After copying these path in output text file, it is done: with printing ‘Success’ out, the program ends.

Below is the C code with detailed commentaries.

#include <stdio.h>

#include <stdlib.h>

typedef struct

{

int row;

int col;

}element;

//a struct for coordinate value in Maze.

typedef struct stack

{

element data;

struct stack \* link;

} Stack;

//To store the path, use Stack implemented by linked List.

int \*\* Maze, \*\* Mark;

/\*Since the size of Maze is unknown, it would be allocated dynamically after checking its size.

Mark is also needed to check whether a certain position is visited or not. in this C code these are global variables.\*/

int Move\_vert[8] = { -1, -1, 0, 1, 1, 1, 0, -1 };

int Move\_horiz[8] = { 0, 1, 1, 1, 0, -1, -1, -1 };

//these are offsets starting from North clockwise, not implemented by struct having two variables but just two arrays.

int FileRead(FILE \* fp1, int \* Start\_Row, int \* Start\_Col, int \* Exit\_Row, int \* Exit\_Col);

void MakeMaze(int Exit\_Row, int Exit\_Col);

void MazePath(int Start\_Row, int Start\_Col, int Exit\_Row, int Exit\_Col, Stack \*\* top);

int RecurPath(int Exit\_Row, int Exit\_Col, Stack \*\* top, int Cur\_Row, int Cur\_Col);

void FreeMaze(int Exit\_Row, int Exit\_Col);

void FileWrite(FILE \* fp2, Stack \*\* top);

Stack \* getnode();

void Push(Stack \*\* top, element data);

element Pop(Stack\*\* top);

//functions for Maze-path finding and stack. Below for more details.

int main(int argc, char \*\* argv) //command-line arguments, where what you write is passed to main function during program's executing.

{

if (argc < 2) //at least 2 text files(Input and Output) are required.

{

printf("Please write : exe Input.txt Output.txt\n");

return 0;

}

int Start\_Row, Start\_Col, Exit\_Row, Exit\_Col;

FILE \* fp1 = NULL;

fopen\_s(&fp1, argv[1], "r"); //with argv[1] read an Input text file. if failed, just return 0.

if (FileRead(fp1, &Start\_Row, &Start\_Col, &Exit\_Row, &Exit\_Col) < 0) return 0;

Stack \* top = NULL; //set top as NULL.

MazePath(Start\_Row, Start\_Col, Exit\_Row, Exit\_Col, &top);

//Path's indexes from exit to start position reversely are pushed in stack. below for more details.

FreeMaze(Exit\_Row, Exit\_Col);

//there's no need to maintain a Maze and Mark so get them be freed.

FILE \* fp2 = NULL;

fopen\_s(&fp2, argv[2], "w"); //with argv[2] write what stack has in an output file.

FileWrite(fp2, &top);

return 0;

}

int FileRead(FILE \* fp1, int \*Start\_Row, int \* Start\_Col, int \* Exit\_Row, int \* Exit\_Col)

{

if (fp1 == NULL)

{

printf("Input.txt fopen fail\n");

fclose(fp1);

return -1;

}

fscanf\_s(fp1, "%d %d %d %d", Start\_Row, Start\_Col, Exit\_Row, Exit\_Col, sizeof(int));

//at first time read first 4 values:Starting position and Exit position.

MakeMaze(\*Exit\_Row, \*Exit\_Col);//then with this function make space.

for (int i = \*Start\_Row; i <= \*Exit\_Row; i++)//Copt the Input text.

for (int j = \*Start\_Col; j <= \*Exit\_Col; j++)

fscanf\_s(fp1, "%d", &Maze[i][j], sizeof(int));

fclose(fp1);

return 0;

}

void MakeMaze(int Exit\_Row, int Exit\_Col)

//Called by FileRead, allocate Maze and also Mark dynamically. Maze's four sides are blocked by 1 for convenience in advance.

{

int i;

Maze = (int \*\*)calloc(Exit\_Row + 2, sizeof(int\*));

Mark = (int \*\*)calloc(Exit\_Row + 2, sizeof(int\*));

for (i = 0; i < (Exit\_Row + 2); i++)

{

Maze[i] = (int \*)calloc(Exit\_Col + 2, sizeof(int));

Mark[i] = (int \*)calloc(Exit\_Col + 2, sizeof(int));

}

for (i = 0; i < (Exit\_Col + 2); i++)

{

Maze[0][i] = 1;

Maze[Exit\_Row + 1][i] = 1;

}

for (i = 1; i < (Exit\_Row + 1); i++)

{

Maze[i][0] = 1;

Maze[i][Exit\_Col + 1] = 1;

}

}

void MazePath(int Start\_Row, int Start\_Col, int Exit\_Row, int Exit\_Col, Stack \*\* top)

{

int Result;

if (Maze[Start\_Row][Start\_Col] == 1) Result = -1;

//you must enter start position at least.

else Result = RecurPath(Exit\_Row, Exit\_Col, &(\*top), Start\_Row, Start\_Col);

//with recursive function RecurPath, find path. if path is found, result becomes 0.

element data;

data.col = Start\_Col;

data.row = Start\_Row;

Push(&(\*top), data);

/\*whether found or not, you should push start position to stack.

if path is found, next position (not current position) is pushed to stack at that moment next position becomes exit.

so even after RecurPath ends, there is no starting position in stack. so you have to push it after executuing it.\*/

return;

}

int RecurPath(int Exit\_Row, int Exit\_Col, Stack \*\* top, int Cur\_Row, int Cur\_Col)

{

element data;

Mark[Cur\_Row][Cur\_Col] = 1;

//mark current position with 1 so that you do not have to enter repeatedly.

if (Cur\_Row == Exit\_Row && Cur\_Col == Exit\_Col) return 0;

//if path is found, at that point return 0 to MazePath.

int Result = -1;

for (int i = 0; i < 8; i++)

{

int Next\_Row = Cur\_Row + Move\_vert[i];

int Next\_Col = Cur\_Col + Move\_horiz[i];

//check from North clockwise 8 times.

if (Mark[Next\_Row][Next\_Col] == 1 || Maze[Next\_Row][Next\_Col] == 1) continue;

//if next position is already visited or blocked, just skip.

Result = RecurPath(Exit\_Row, Exit\_Col, &(\*top), Next\_Row, Next\_Col);

//check whether next position is exit or not-path is found or not from Result's value.

if (Result == 0)

/\*if Result becomes 0 then path is found, at that moment push its next position to stack. Since push is called at that time its next step becomes Exit, path's coordinate values are stored in stack reversely from exit to start.\*/

{

data.col = Next\_Col;

data.row = Next\_Row;

Push(&(\*top), data);

break;

}

}

if (Result != 0) Mark[Cur\_Row][Cur\_Col] = 0; //unmark current position in case another path could be found.

return Result; //if path is found, it becomes 0 otherwise -1. a key value of this function.

}

void FreeMaze(int Exit\_Row, int Exit\_Col) //these two dimensional arrays Maze and Mark are freed step by step.

{

int i;

for (i = 0; i < Exit\_Row + 2; i++)

{

free(Maze[i]);

free(Mark[i]);

}

free(Maze);

free(Mark);

}

Stack \* getnode() //detached from push, make a new place for data and return it.

{

Stack \* tmp = (Stack\*)malloc(sizeof(Stack));

tmp->link = NULL;

return tmp;

}

void Push(Stack \*\* top, element data)

//make a new place for pushed data by calling getnode. pointing this new node, top is updated and connected to previous data.

{

Stack \* tmp = \*top;

\*top = getnode();

(\*top)->data = data;

(\*top)->link = tmp;

}

element Pop(Stack\*\* top) //if data is in stack, remove and return it. top is also renewed.

{

if (\*top == NULL) exit (0);

Stack \* tmp = \*top;

element data = tmp->data;

\*top = tmp->link;

free(tmp);

return data;

}

void FileWrite(FILE \* fp2, Stack \*\* top)

{

if (fp2 == NULL)

{

printf("Output.txt fopen fail\n");

fclose(fp2);

return;

}

while (\*top)

/\*until there's no data in stack, take it out in Output file. since coordinate values are stored 'from Exit to Start'

at the outset it is naturally poped 'from Start to Exit' out of top.\*/

{

element data = Pop(&(\*top));

fprintf(fp2, "%d %d\n", data.row, data.col);

}

printf("Success\n");

fclose(fp2);

return;

}