Amortization Table

Rachel Hamilton, Jake Cousino

Wednesday, June 20

Contents

Rubric

25 pts commented and readable code ________________

25 pts elegant source code ________________

25 pts concise design ________________

25 pts results during testing ________________

Part I
Source Code

mazeSolver.cpp

```cpp
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <string>
#include <vector>

using namespace std;


int backtrackingMazeSolver(int i, int j);
int greedyMazeSolver(int i, int j, int endX, int endY);
std::vector<int> GetClosestNodeToFinish(int i, int j, int endX, int endY);
int divideAndConquerMazeSolver(int i, int j);
int dynamicProgrammingMazeSolver(int i, int j);
int randomizedMazeSolver(int i, int j);
bool isEmpty(int i, int j);
bool hasBeenChecked(int i, int j);
int bruteForceMazeSolver(int i, int j, bool oneShot, int startX, int startY);
void printArray(struct maze myMaze);
bool bruteCheckForEmpty(int i, int j);
bool bruteCheckForTraveled(int i, int j);
bool isFinishAdjacent(int i, int j);

struct maze
{
    int rows;
    int cols;
                int startX;
                int startY;
    char matrix [100][100];
};

maze myMaze;
int bruteForceCount = 0;

int main()
{
    //required variables
    ifstream in;
    in.open(”maze.txt”);
    char line;

    //read the matrix using plain c code, character by character
    in >> myMaze.rows;
    in >> line;
    in >> myMaze.cols;
    cout << ”Reading␣a␣” << myMaze.rows << ”␣by␣” << myMaze.cols << ”␣matrix.” << endl;
    //Burn the end of line character
    in.ignore(200,’\n’);
    for(int i=0; i<myMaze.rows; i++)
    {
        for(int j=0; j<myMaze.cols; j++)
        {
            in.get( myMaze.matrix[i][j] );
        }
        //Burn the end of line character
        in.ignore(200,’\n’);
    }

    //Print the empty maze
    for(int i=0; i<myMaze.rows; i++)
    {
        for(int j=0; j<myMaze.cols; j++)
            cout << myMaze.matrix[i][j];
        cout << endl;
    }
    int x=1,y=1;
    int endX=1,endY=1;

    //Find starting coordinates
    for(int i=0; i<myMaze.rows; i++)
        for(int j=0; j<myMaze.cols; j++)
            if( myMaze.matrix[i][j] == 'S' ){
                x=i;
                y=j;
                                                                myMaze.startX = x;
                                                                myMaze.startY = y;
            }

    // Find Finish coordinates
    // for(int i=0; i<myMaze.rows; i++)
    // for(int j=0; j<myMaze.cols; j++)
    // if( myMaze.matrix[i][j] == 'F' ){
    //     endX=j;
    //     endY=i;
    // }

    //Call a recursive mazeSolver
    //FIXME:RH:int bfDistance = bruteForceMazeSolver(x,y);      //brute force? dnc?
    //int btDistance = backtrackingMazeSolver(x,y);      //brute force? dnc?
    // int gDistance = greedyMazeSolver(x,y,endX,endY);
    int dncDistance = divideAndConquerMazeSolver(x,y);
    int dpDistance = dynamicProgrammingMazeSolver(x,y);
    int rDistance = randomizedMazeSolver(x,y);

    //cout << "Brute force distance: " << bfDistance << " units away!" << endl;
    //cout << "Backtracking distance: " << btDistance << " units away!" << endl;
    // cout << "Greedy distance: " << gDistance << " units away!" << endl;
    cout << "Divide␣and␣conquer␣distance:␣" << dncDistance << "␣units␣away!" << endl;
    cout << "Dynamic␣programming␣distance:␣" << dpDistance << "␣units␣away!" << endl;
    cout << "Randomized␣distance:␣" << rDistance << "␣units␣away!" << endl;

    //Print solved maze - x2
    // for(int i=0; i<myMaze.rows; i++)
    // {
    //     for(int j=0; j<myMaze.cols; j++)
    //     cout << myMaze.matrix[i][j];
    //     cout << endl;
    // }

    // ***************************************
    // Begin Student Written Section
    // ***************************************


    // char **mazeArray = (char**)malloc(myMaze.rows * sizeof(char*));
    // for(int i=0 ; i<myMaze.rows ; i++){
    //     mazeArray[i] = (char*)malloc(myMaze.cols * sizeof(char*));
    // }


    bruteForceMazeSolver(1, 1, false, x, y);


    return 0;
}

int bruteForceMazeSolver(int i, int j, bool oneShot, int startX, int startY)
{
    bool isEmptySpace, isTraveledSpace;

    bruteForceCount++;

    // will not activate unless this is the first iteration
    // gives starting location
    if(!oneShot){
        i = startX;
        j = startY;
        oneShot = true;
        myMaze.matrix[i][j] = '.';
    }

    // Check if adjacent spot is F
    if(isFinishAdjacent(i,j)){
        myMaze.matrix[i][j] = '.';
                                myMaze.matrix[myMaze.startX][myMaze.startY] = 'S';
                                cout << endl << "Brute␣Force:" << endl;
                                printArray(myMaze);
        cout << "Brute␣distance:␣" << bruteForceCount << endl;
    }
    else{
        // if a finish spot is not nearby, check for empty space
        isEmptySpace = bruteCheckForEmpty(i,j);
        if(!isEmptySpace){
            // if there are no empty spaces, check for already traveled spaces
            isTraveledSpace = bruteCheckForTraveled(i,j);
            if(!isTraveledSpace){
                cout << "ERROR";
            }
        }
    }

    return -1;
}
int backtrackingMazeSolver(int i, int j)
{
    //algorithm goes here
    return -1;
}
// int greedyMazeSolver(int i, int j, int endX, int endY)
// {
//     if(myMaze.matrix[i][j] == 'F') return 1;
//
//     std::vector<int> nextNode = GetClosestNodeToFinish(i, j, endX, endY);
//     myMaze.matrix[nextNode[0]][nextNode[1]] = '@';
//     return greedyMazeSolver(nextNode[0], nextNode[1], endX, endY);
//
//   return -1;
// }
//
// std::vector<int> GetClosestNodeToFinish(int i, int j, int endX, int endY)
// {
//     std::vector<int> north = {i, j-1};
//     std::vector<int> east = {i+1, j};
//     std::vector<int> south = {i, j+1};
//     std::vector<int> west = {i-1, j};
//
//     std::vector<std::vector<int>> directions;
//     directions.push_back(north);
//     directions.push_back(east);
//     directions.push_back(south);
//     directions.push_back(west);
//
//     std::vector<int> clostestNode = south;
//     for(int i = 0; i < 4; i++)
//     {
```

```
//         int nodeDistance = std::abs(clostestNode[0] - endX) + std::abs(clostestNode[1] - endY);
//         if(nodeDistance > (std::abs(directions[i][0] - endX) + std::abs(directions[i][1] - endY)))
//         {
//             if(myMaze.matrix[directions[i][0]][directions[i][1]] != '*')
//             {
//                 clostestNode = directions[i];
//             }
//         }
//     }
//     return clostestNode;
// }
int divideAndConquerMazeSolver(int i, int j)
{
    //algorithm goes here
    return -1;
}
int dynamicProgrammingMazeSolver(int i, int j)
{
    //algorithm goes here
    return -1;
}
int randomizedMazeSolver(int i, int j)
{
    //algorithm goes here
    return -1;
}

// Added by RH
// given the maze, and the space to check, will check
// for to see if the lacation is valid first, then if so
// will check the character stored is a space character
// (not in the Master Chief sorta way), then set the return
// condition to true
bool isEmpty(int i, int j){
    bool isEmpty = false;
    bool isValidLocation = true;

    // check if valid
    if(myMaze.rows < i){
        cout << "Not␣a␣valid␣row␣lacation";
        isValidLocation = !isValidLocation;
    }
    if(myMaze.cols < j){
        cout << "Not␣a␣valid␣col␣lacation";
        isValidLocation = !isValidLocation;
    }

    // check if space
    if(isValidLocation){
        if(myMaze.matrix[i][j] == '␣')
        {
            isEmpty = true;
        }
    }

    return isEmpty;

}

// Added by RH
// Same as isValid, but for check the char '.'
bool hasBeenChecked(int i, int j){
    bool isTracked = false;
    bool isValidLocation = true;

    // check if valid
    if(myMaze.rows < i){
        cout << "Not␣a␣valid␣row␣lacation";
        isValidLocation = !isValidLocation;
    }
    if(myMaze.cols < j){
        cout << "Not␣a␣valid␣col␣lacation";
        isValidLocation = !isValidLocation;
    }


    // check if space
    if(isValidLocation){
        if(myMaze.matrix[i][j] == '.'){
            isTracked = true;
        }
    }

    return isTracked;

}

void printArray(struct maze array){
    for(int i=0 ; i<myMaze.rows ; i++){
        for(int j=0 ; j<myMaze.cols ; j++){
            cout << array.matrix[i][j];
        }
        cout << endl;
    }
}


// Checks for empty spots in direction order east, north, south, west
// Marks a '.' on every traveled spot
bool bruteCheckForEmpty(int i, int j){
    if(isEmpty(i,j+1)){
        myMaze.matrix[i][j] = '.';
        bruteForceMazeSolver(i, j+1, true, 0, 0);
    }
    else{
        if(isEmpty(i+1,j)){
            myMaze.matrix[i][j] = '.';
            bruteForceMazeSolver(i+1, j, true, 0, 0);
        }
        else{
            if(isEmpty(i-1,j)){
                myMaze.matrix[i][j] = '.';
                bruteForceMazeSolver(i-1, j, true, 0, 0);
            }
            else{
                if(isEmpty(i,j-1)){
                    myMaze.matrix[i][j] = '.';
                    bruteForceMazeSolver(i, j-1, true, 0, 0);
                }
                else{
                    return false;
                }
            }
        }
    }
    return true;
}


// Runs the same thing as bruteCheckForEmpty, except places '@'
bool bruteCheckForTraveled(int i, int j){
    //if(!isFinishAdjacent(myMaze,i,j)){
        if(hasBeenChecked(i,j+1)){
            myMaze.matrix[i][j] = '@';
            bruteForceMazeSolver(i, j+1,true, 0, 0);
        }
        else{
            if(hasBeenChecked(i+1,j)){
                myMaze.matrix[i][j] = '@';
                bruteForceMazeSolver(i+1, j, true, 0, 0);
            }
            else{
                if(hasBeenChecked(i-1,j)){
                    myMaze.matrix[i][j] = '@';
                    bruteForceMazeSolver(i-1, j, true, 0, 0);
                }
                else{
                    if(hasBeenChecked(i,j-1)){
                        myMaze.matrix[i][j] = '@';
                        bruteForceMazeSolver(i, j-1, true, 0, 0);
                    }
                    else{
                        return false;
                    }
                }
            }
        }
    //}
    return true;
}

// checks all directions for adjacent 'F'
bool isFinishAdjacent(int i, int j){


    if(myMaze.matrix[i+1][j]=='F'){
        return true;
    }
    if(myMaze.matrix[i-1][j]=='F'){
        return true;
    }
    if(myMaze.matrix[i][j+1]=='F'){
        return true;
    }
    if(myMaze.matrix[i][j-1]=='F'){
        return true;
    }

    return false;
}

//recursion!!
//Mark current location
//Base Case: Look north, south, east, west for victory!
//Mark our path
//Try going south if it is open
//Try going north if it is open
//Try going east if it is open
//Try going west if it is open
```

Part II

Output

```
Reading a 20 by 20 matrix.
********************
*  F*        *   * *
* *** ****** * *   *
*   * *      * *****
*** * ********     *
* * *          *** *
* * ********** *S* *
* *      *   * * * *
* ****** * * * * * *
*      * * * * *   *
****** * * * * *****
*    * * * * * *   *
* ** * * * * * * * *
* **** * * *   * * *
* *      * *** *** *
* * ******   *     *
* *          ***** *
* **************** *
*                  *
********************
Divide and conquer distance: -1 units away!
Dynamic programming distance: -1 units away!
Randomized distance: -1 units away!

Brute Force:
********************
*..F*        *   * *
*.*** ****** * *   *
*...* *      * *****
***.* ********.....*
* *.*         .***.*
* *.**********.*S*.*
* *......*...*.*.*.*
* ******.*.*.*.*.*.*
*      *.*.*.*.*...*
****** *.*.*.*.*****
*@@@@* *.*.*.*.*@@@*
*@**@* *.*.*.*.*@*@*
*@**** *.*.*...*@*@*
*@*......*.***@***@*
*@*.******...*@@@@@*
*@*..........*****@*
*@****************@*
*@@@@@@@@@@@@@@@@@@*
********************
Brute distance: 172
```