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

```
#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++)
```

```
62      {
63          for(int j=0; j<myMaze.cols; j++)
64              cout << myMaze.matrix[i][j];
65          cout << endl;
66      }
67      int x=1,y=1;
68      int endX=1,endY=1;
69
70      //Find starting coordinates
71      for(int i=0; i<myMaze.rows; i++)
72          for(int j=0; j<myMaze.cols; j++)
73              if( myMaze.matrix[i][j] == 'S' ){
74                  x=i;
75                  y=j;
76                                                          myMaze.startX = x;
77                                                          myMaze.startY = y;
78              }
79
80      // Find Finish coordinates
81      // for(int i=0; i<myMaze.rows; i++)
82      // for(int j=0; j<myMaze.cols; j++)
83      // if( myMaze.matrix[i][j] == 'F' ){
84      //     endX=j;
85      //     endY=i;
86      // }
87
88      //Call a recursive mazeSolver
89      //FIXME:RH:int bfDistance = bruteForceMazeSolver(x,y);      //brute force? dnc?
90      //int btDistance = backtrackingMazeSolver(x,y);      //brute force? dnc?
91      // int gDistance = greedyMazeSolver(x,y,endX,endY);
92      int dncDistance = divideAndConquerMazeSolver(x,y);
93      int dpDistance = dynamicProgrammingMazeSolver(x,y);
94      int rDistance = randomizedMazeSolver(x,y);
95
96      //cout << "Brute force distance: " << bfDistance << " units away!" << endl;
97      //cout << "Backtracking distance: " << btDistance << " units away!" << endl;
98      // cout << "Greedy distance: " << gDistance << " units away!" << endl;
99      cout << "Divide and conquer distance: " << dncDistance << " units away!" << endl;
100     cout << "Dynamic programming distance: " << dpDistance << " units away!" << endl;
101     cout << "Randomized distance: " << rDistance << " units away!" << endl;
102
103     //Print solved maze - x2
104     // for(int i=0; i<myMaze.rows; i++)
105     // {
106     //     for(int j=0; j<myMaze.cols; j++)
107     //     cout << myMaze.matrix[i][j];
108     //     cout << endl;
109     // }
110
111     // ************************************
112     // Begin Student Written Section
113     // ************************************
114
115
116     // char **mazeArray = (char**)malloc(myMaze.rows * sizeof(char*));
117     // for(int i=0 ; i<myMaze.rows ; i++){
118     //     mazeArray[i] = (char*)malloc(myMaze.cols * sizeof(char*));
119     // }
120
121
122     bruteForceMazeSolver(1, 1, false, x, y);
123
124
125     return 0;
126 }
127
128 int bruteForceMazeSolver(int i, int j, bool oneShot, int startX, int startY)
129 {
```

```
130     bool isEmptySpace, isTraveledSpace;
131
132     bruteForceCount++;
133
134     // will not activate unless this is the first iteration
135     // gives starting location
136     if(!oneShot){
137         i = startX;
138         j = startY;
139         oneShot = true;
140         myMaze.matrix[i][j] = '.';
141     }
142
143     cout << endl << endl;
144
145
146     // Check if adjacent spot is F
147     if(isFinishAdjacent(i,j)){
148         myMaze.matrix[i][j] = '.';
149                                 myMaze.matrix[myMaze.startX][myMaze.startY] = 'S';
150                                 printArray(myMaze);
151         cout << "Brute distance: " << bruteForceCount << endl;
152     }
153     else{
154         // if a finish spot is not nearby, check for empty space
155         isEmptySpace = bruteCheckForEmpty(i,j);
156         if(!isEmptySpace){
157             // if there are no empty spaces, check for already traveled spaces
158             isTraveledSpace = bruteCheckForTraveled(i,j);
159             if(!isTraveledSpace){
160                 cout << "ERROR";
161             }
162         }
163     }
164
165     return -1;
166 }
167 int backtrackingMazeSolver(int i, int j)
168 {
169     //algorithm goes here
170     return -1;
171 }
172 // int greedyMazeSolver(int i, int j, int endX, int endY)
173 // {
174 //     if(myMaze.matrix[i][j] == 'F') return 1;
175 //
176 //     std::vector<int> nextNode = GetClosestNodeToFinish(i, j, endX, endY);
177 //     myMaze.matrix[nextNode[0]][nextNode[1]] = '@';
178 //     return greedyMazeSolver(nextNode[0], nextNode[1], endX, endY);
179 //
180 //   return -1;
181 // }
182 //
183 // std::vector<int> GetClosestNodeToFinish(int i, int j, int endX, int endY)
184 // {
185 //     std::vector<int> north = {i, j-1};
186 //     std::vector<int> east = {i+1, j};
187 //     std::vector<int> south = {i, j+1};
188 //     std::vector<int> west = {i-1, j};
189 //
190 //     std::vector<std::vector<int>> directions;
191 //     directions.push_back(north);
192 //     directions.push_back(east);
193 //     directions.push_back(south);
194 //     directions.push_back(west);
195 //
196 //     std::vector<int> clostestNode = south;
197 //     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;

```

```
264     // check if valid
265     if(myMaze.rows < i){
266         cout << "Not a valid row lacation";
267         isValidLocation = !isValidLocation;
268     }
269     if(myMaze.cols < j){
270         cout << "Not a valid col lacation";
271         isValidLocation = !isValidLocation;
272     }
273
274
275     // check if space
276     if(isValidLocation){
277         if(myMaze.matrix[i][j] == '.'){
278             isTracked = true;
279         }
280     }
281
282     return isTracked;
283
284 }
285
286 void printArray(struct maze array){
287     for(int i=0 ; i<myMaze.rows ; i++){
288         for(int j=0 ; j<myMaze.cols ; j++){
289             cout << array.matrix[i][j];
290         }
291         cout << endl;
292     }
293 }
294
295
296 // Checks for empty spots in direction order east, north, south, west
297 // Marks a '.' on every traveled spot
298 bool bruteCheckForEmpty(int i, int j){
299     if(isEmpty(i,j+1)){
300         myMaze.matrix[i][j] = '.';
301         bruteForceMazeSolver(i, j+1, true, 0, 0);
302     }
303     else{
304         if(isEmpty(i+1,j)){
305             myMaze.matrix[i][j] = '.';
306             bruteForceMazeSolver(i+1, j, true, 0, 0);
307         }
308         else{
309             if(isEmpty(i-1,j)){
310                 myMaze.matrix[i][j] = '.';
311                 bruteForceMazeSolver(i-1, j, true, 0, 0);
312             }
313             else{
314                 if(isEmpty(i,j-1)){
315                     myMaze.matrix[i][j] = '.';
316                     bruteForceMazeSolver(i, j-1, true, 0, 0);
317                 }
318                 else{
319                     return false;
320                 }
321             }
322         }
323     }
324     return true;
325 }
326
327
328 // Runs the same thing as bruteCheckForEmpty, except places '@' and
329 // does not run if the finish is adjacent. Without the check, the program
330 // defaults to looking for '.' spaces rather than stopping
331 bool bruteCheckForTraveled(int i, int j){
```

```
332     //if(!isFinishAdjacent(myMaze,i,j)){
333         if(hasBeenChecked(i,j+1)){
334             myMaze.matrix[i][j] = '@';
335             cout << "east " << i << "x" << j << endl;
336             bruteForceMazeSolver(i, j+1,true, 0, 0);
337         }
338         else{
339             if(hasBeenChecked(i+1,j)){
340                 myMaze.matrix[i][j] = '@';
341                 cout << "north " << i << "x" << j << endl;
342                 bruteForceMazeSolver(i+1, j, true, 0, 0);
343             }
344             else{
345                 if(hasBeenChecked(i-1,j)){
346                     myMaze.matrix[i][j] = '@';
347                     cout << "south " << i << "x" << j  << endl;
348                     bruteForceMazeSolver(i-1, j, true, 0, 0);
349                 }
350                 else{
351                     if(hasBeenChecked(i,j-1)){
352                         myMaze.matrix[i][j] = '@';
353                         cout << "west" << i << "x" << j  << endl;
354                         bruteForceMazeSolver(i, j-1, true, 0, 0);
355                     }
356                     else{
357                         return false;
358                     }
359                 }
360             }
361         }
362     //}
363     return true;
364 }
365 
366 // checks all directions for adjacent 'F'
367 bool isFinishAdjacent(int i, int j){
368 
369 
370     if(myMaze.matrix[i+1][j]=='F'){
371         return true;
372     }
373     if(myMaze.matrix[i-1][j]=='F'){
374         return true;
375     }
376     if(myMaze.matrix[i][j+1]=='F'){
377         return true;
378     }
379     if(myMaze.matrix[i][j-1]=='F'){
380         return true;
381     }
382 
383     return false;
384 }
385 
386 //recursion!!
387 //Mark current location
388 //Base Case: Look north, south, east, west for victory!
389 //Mark our path
390 //Try going south if it is open
391 //Try going north if it is open
392 //Try going east if it is open
393 //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!
```

```
south 12x4

west11x4

west11x3

west11x2

north 11x1

north 12x1

north 13x1

north 14x1

north 15x1

north 16x1

north 17x1

east 18x1

east 18x2

east 18x3

east 18x4

east 18x5

east 18x6

east 18x7
```

```
east 18x8

east 18x9

east 18x10

east 18x11

east 18x12

east 18x13

east 18x14

east 18x15

east 18x16

east 18x17

south 18x18

south 17x18

south 16x18

south 13x16

south 12x16

east 11x16

east 11x17
```

```
north 11x18

north 12x18

north 13x18

north 14x18

west15x18

west15x17

west15x16

west15x15

south 15x14

south 14x14
```

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