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     // Check if adjacent spot is F
144     if(isFinishAdjacent(i,j)){
145         myMaze.matrix[i][j] = '.';
146                                 myMaze.matrix[myMaze.startX][myMaze.startY] = 'S';
147                                 cout << endl << "Brute Force:" << endl;
148                                 printArray(myMaze);
149         cout << "Brute distance: " << bruteForceCount << endl;
150     }
151     else{
152         // if a finish spot is not nearby, check for empty space
153         isEmptySpace = bruteCheckForEmpty(i,j);
154         if(!isEmptySpace){
155             // if there are no empty spaces, check for already traveled spaces
156             isTraveledSpace = bruteCheckForTraveled(i,j);
157             if(!isTraveledSpace){
158                 cout << "ERROR";
159             }
160         }
161     }
162
163     return -1;
164 }
165 int backtrackingMazeSolver(int i, int j)
166 {
167     //algorithm goes here
168     return -1;
169 }
170 // int greedyMazeSolver(int i, int j, int endX, int endY)
171 // {
172 //     if(myMaze.matrix[i][j] == 'F') return 1;
173 //
174 //     std::vector<int> nextNode = GetClosestNodeToFinish(i, j, endX, endY);
175 //     myMaze.matrix[nextNode[0]][nextNode[1]] = '@';
176 //     return greedyMazeSolver(nextNode[0], nextNode[1], endX, endY);
177 //
178 //   return -1;
179 // }
180 //
181 // std::vector<int> GetClosestNodeToFinish(int i, int j, int endX, int endY)
182 // {
183 //     std::vector<int> north = {i, j-1};
184 //     std::vector<int> east = {i+1, j};
185 //     std::vector<int> south = {i, j+1};
186 //     std::vector<int> west = {i-1, j};
187 //
188 //     std::vector<std::vector<int>> directions;
189 //     directions.push_back(north);
190 //     directions.push_back(east);
191 //     directions.push_back(south);
192 //     directions.push_back(west);
193 //
194 //     std::vector<int> clostestNode = south;
195 //     for(int i = 0; i < 4; i++)
196 //     {
```

```
197 //          int nodeDistance = std::abs(clostestNode[0] - endX) + std::abs(clostestNode[1] -
        endY);
198 //          if(nodeDistance > (std::abs(directions[i][0] - endX) + std::abs(directions[i][1]
        - endY)))
199 //          {
200 //              if(myMaze.matrix[directions[i][0]][directions[i][1]] != '*')
201 //              {
202 //                  clostestNode = directions[i];
203 //              }
204 //          }
205 //      }
206 //      return clostestNode;
207 // }
208 int divideAndConquerMazeSolver(int i, int j)
209 {
210     //algorithm goes here
211     return -1;
212 }
213 int dynamicProgrammingMazeSolver(int i, int j)
214 {
215     //algorithm goes here
216     return -1;
217 }
218 int randomizedMazeSolver(int i, int j)
219 {
220     //algorithm goes here
221     return -1;
222 }
223
224 // Added by RH
225 // given the maze, and the space to check, will check
226 // for to see if the lacation is valid first, then if so
227 // will check the character stored is a space character
228 // (not in the Master Chief sorta way), then set the return
229 // condition to true
230 bool isEmpty(int i, int j){
231     bool isEmpty = false;
232     bool isValidLocation = true;
233
234     // check if valid
235     if(myMaze.rows < i){
236         cout << "Not a valid row lacation";
237         isValidLocation = !isValidLocation;
238     }
239     if(myMaze.cols < j){
240         cout << "Not a valid col lacation";
241         isValidLocation = !isValidLocation;
242     }
243
244     // check if space
245     if(isValidLocation){
246         if(myMaze.matrix[i][j] == ' ')
247         {
248             isEmpty = true;
249         }
250     }
251
252     return isEmpty;
253
254 }
255
256 // Added by RH
257 // Same as isValid, but for check the char '.'
258 bool hasBeenChecked(int i, int j){
259     bool isTracked = false;
260     bool isValidLocation = true;
261
262     // check if valid
```

```
263     if(myMaze.rows < i){
264         cout << "Not a valid row lacation";
265         isValidLocation = !isValidLocation;
266     }
267     if(myMaze.cols < j){
268         cout << "Not a valid col lacation";
269         isValidLocation = !isValidLocation;
270     }
271
272
273     // check if space
274     if(isValidLocation){
275         if(myMaze.matrix[i][j] == '.'){
276             isTracked = true;
277         }
278     }
279
280     return isTracked;
281
282 }
283
284 void printArray(struct maze array){
285     for(int i=0 ; i<myMaze.rows ; i++){
286         for(int j=0 ; j<myMaze.cols ; j++){
287             cout << array.matrix[i][j];
288         }
289         cout << endl;
290     }
291 }
292
293
294 // Checks for empty spots in direction order east, north, south, west
295 // Marks a '.' on every traveled spot
296 bool bruteCheckForEmpty(int i, int j){
297     if(isEmpty(i,j+1)){
298         myMaze.matrix[i][j] = '.';
299         bruteForceMazeSolver(i, j+1, true, 0, 0);
300     }
301     else{
302         if(isEmpty(i+1,j)){
303             myMaze.matrix[i][j] = '.';
304             bruteForceMazeSolver(i+1, j, true, 0, 0);
305         }
306         else{
307             if(isEmpty(i-1,j)){
308                 myMaze.matrix[i][j] = '.';
309                 bruteForceMazeSolver(i-1, j, true, 0, 0);
310             }
311             else{
312                 if(isEmpty(i,j-1)){
313                     myMaze.matrix[i][j] = '.';
314                     bruteForceMazeSolver(i, j-1, true, 0, 0);
315                 }
316                 else{
317                     return false;
318                 }
319             }
320         }
321     }
322     return true;
323 }
324
325
326 // Runs the same thing as bruteCheckForEmpty, except places '@'
327 bool bruteCheckForTraveled(int i, int j){
328     //if(!isFinishAdjacent(myMaze,i,j)){
329         if(hasBeenChecked(i,j+1)){
330             myMaze.matrix[i][j] = '@';
```

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