Samit Basnet

CPS112: Computational Thinking with Algorithms and Data Structures

Homework 4

Part 7: Complexity Analysis

*Worst-case analysis of SolveMaze in terms of n(number of locations in the maze(width*height)*

```
        public ArrayList solveMaze(Maze maze, MazeGUI graphics)

{

1       MazeGUI gui=graphics;
2       moves.clear();
3       int row=maze.getNumRows();
4       int col=maze.getNumCols();
5       GridLocation start=maze.getStartLocation();
6       GridLocation goal=maze.getGoalLocation();
7       moves.addLocation(start);
8       visited.add(start);
9       added.add(start);
10      addedby.add(start);
11      current=start;

12      while(!current.equals(goal)) {
13              if ( current.equals(start)){
14                      moves.clear();   }
15              int currentrow=current.getRow();
16              int currentcol=current.getColumn();

#For North

17              if(maze.isValidLoc((currentrow-1),currentcol))  {
18                      GridLocation loca=new   GridLocation(currentrow-1,currentcol);
19                      GridLocation location=maze.getMazeLocation(loca);

20                      if(maze.getSquare(location)!='#'&& visited.contains(location)!=true) {
                                        moves.addLocation(location);
                                        gui.addLocToAgenda(location);
                                        addedby.add(current);
                                        added.add(location);  }}

#Same for South, East and West except
                South= if(maze.isValidLoc((currentrow+1),currentcol))  {
                East=   if(maze.isValidLoc(currentrow,currentcol+1))  {
                West=  if(maze.isValidLoc(currentrow,currentcol-1))  {
                [in the same order]
```

```
            21      if(moves.isEmpty()) {
                            return path;   }

            22      current=moves.getLocation();
                    gui.pause(1);
                    gui.visitLoc(current);
                    visited.add(current);

    }  //End of while loop

23  GridLocation cur=goal;
24  path.add(goal);

25  while (!cur.equals(start))          {

            int idx=added.indexOf(cur);
            path.add(addedby.get(idx));
            cur=addedby.get(idx);

       }

26  for(int j=path.size()-1; j>=0;j--)

       {

            sortedpath.add(path.get(j));
            gui.addLocToPath(path.get(j));

       }

27  return sortedpath;

  } //End of SolveMaze
```

Example input file: mazefile3.txt

3 5

#####
**.###*
#o…

Using a stackagenda(LIFO)

1. Step-by-step execution

Lines 1-4: MazeGUI gui=graphics; moves.clear(); int row=3; int col=5;

Lines 5-6: GridLocation start= (2,1); GridLocation goal=(1,0);

Lines 7-11: moves.addLocation(2,1); visited.add(2,1); added.add(2,1); addedby.add(2,1);

current=(2,1);

#While loop

First Iteration

Lines 12-16: (2,1) not equals goal=(1,0), so moves to the next line

(2,1) equals start, so stackagenda moves gets cleared—moves is empty

int currentRow=2; int currentcol=current.getColumn();

Check for North: maze.isValid((2-1),1): True—(1,1)!= "#" and visited.contains(1,1)!=: True

Adds (1,1) to moves, added, visited and adds (2,1) to addedby

Check for South: (3,1) not valid

Check for East: maze.isValid(2,2): True—(2,2)!= "#" and visited.contains(2,2)!=: True

Adds (2,2) to moves, added, visited and adds(1,1) to addedby

Check for West: maze.isValid((2,0): True—(2,0)!= "#" and visited.contains(1,1)!=: False

Lines 21-22: moves is not empty so doesn't return path

current= moves.getLocation();=(2,2) as moves is a stack (1,1) is at the bottom and (2,2) on top of it.

Second Iteration: current=(2,2)

North(1,2): invalid

South(3,2): invalid

East(2,3): gets added to moves, added, visited and (2,2) gets added to addedby

West(2,1): already in visited so invalid

Third Iteration: current=(2,3)

North(1,3): invalid

South(3,3): invalid

East(2,4): gets added to the moves, added, visited and (2,3) gets added to addedby

West(2,2): already in visited so invalid

Fourth Iteration: current=(2,4)

North, South, East, West: all invalid

current=moves.getLocation()=(1,1);

Fifth Iteration: current=(1,1)

North: (0,1) is invalid

South: (2,0) is invalid as visited already has the start location

East: invalid as it has a wall

West: (1,0) gets added to moves, added, visited and (1,1) gets added to addedby

Current=moves.getLocation()=(1,0);

Sixth Iteration: current(1,0)=goal so the loop ends

Lines 23-25: cur=goal=(1,0);

added=[(2,1), (1,1), (2,2),(2,3),(2,4),(1,0)]

addedby=[(2,1),(2,1),(2,1),(2,2),(2,3),(1,1)]

path.add(1,0);

While loop

int idx=added.indexOf(1,0);= 5

path.add(addedby.get(5));=(1,1)

cur=(1,1)

Similarly, (1,1) was added by (2,1) so (2,1) gets added to path which is start so loop ends

Line 26-27: *Forloop*

Path=[(1,0),(1,1),(2,1)];

Runs until the length of path and reverses the order of contents from start to goal to give sorted path=[(2,1),(1,1),(1,0)] and returns the sorted path.

Description: This function returns the shortest path from start to goal of a maze if a solution can be found and returns an empty arraylist path if the maze cannot be solved.

2. Analysis

Lets consider an example input mazefile3.txt which has 3 rows and 5 columns. The lines 1-4 of solveMaze initializes MazeGUI, clears the stackagenda and gets the row and column of the maze, so it can be considered as O(1). Lines 5-11 are all just assignment and getting locations so it can also be considered as constant time O(1). Line 12-22 is a while loop that runs until the current location is not equals to goal. It has 4 if statements which has 5 instructions each, collectively we can say that the complexity of the loop is O(n) as it depends on the number of rows and columns of the maze file. Line 25 is a while loop that finds the path from start to goal which depends on the size of the visited arraylist, it also depends on the rows and columns of the given maze, so it can be said to have O(n) complexity. Lastly, we have a forloop that sorts the path from start to goal, so it is also O(n) complexity.

In overall, the complexity is O(n)+O(n)+O(n)=3O(n). As the overall complexity only depends on the number of rows and columns of the maze file and the loops the file has, we can say that it is O(n) as the loops aren't nested.

Overall Time Complexity: O(n)

A. What properties of mazes are more important in determining which one is more efficient? In what kinds of mazes is the queue-based agenda more efficient and in what kinds is the stack-based agenda more efficient?

The properties of maze important in determining which one is more efficient are the number of walls and how they are arranged. In other words, the presence of narrow paths or an open space with few walls can determine which agenda will be favored.

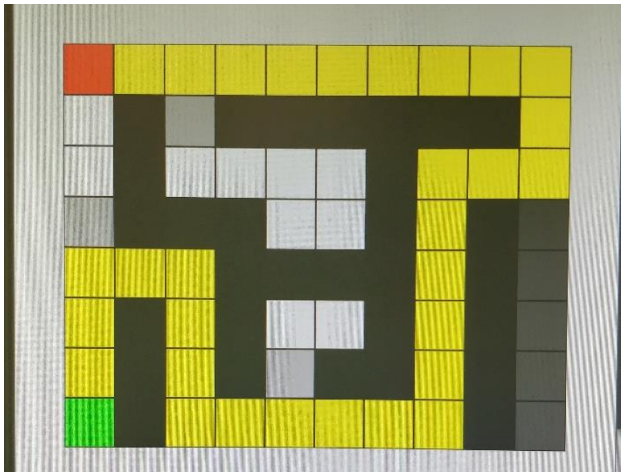
A stack-based agenda is more efficient in an wide open maze which doesn't have narrow paths and less walls. The way that stack-based agenda works is that it moves linearly and vertically so when the maze is open without many walls, it is easier for it to find the goal as it moves in a linear or vertical fashion.

A queue-based agenda is more efficient in case of mazes having a narrow path with wall as boundaries around them. As the queue-based agenda expands in a circular motion in all directions, it is easier for it to find a goal in case of narrow paths, specially when the goal is in the immediate radius.

B. Quality of solutions. Does one agenda always find a shorter path from start to goal? If so, explain why that one is always best.
If not, describe properties of maze that make one or the other agenda produce a better solution. Supply at least one example maze that helps illustrate your points and demonstrate the differences between the two algorithms.

Queue based maze always find a shorter path from start to goal or equal to the one given by stack. As we discussed, this is favored depends on the type of maze, open or with narrow paths.

Lets consider a maze having narrow paths with many walls.



This image alongside is the execution of the same maze using a stack-based agenda. A stack based agenda moves linearly or vertically which gives us the following path when the goal is towards the north.

The second image is the result of execution using a queue-based agenda. The queue-based agenda expands radially in all direction, so it can cross through the walls. This approach is advantageous when you have a maze with narrow path and lots of walls. The stack follows linear path but queue expands radially so can find a shorter path wherever it be.

So in these type of case, queue-based agenda is more efficient.