Project 3: Rat in a Maze Problem

Documentation

Project description:

The problem at hand involves modeling a maze using a matrix of dimensions N * N. The maze is represented as a grid, where each cell in the matrix corresponds to a location within the maze. The upper left cell is designated as the starting point, and the lower right cell is the target or endpoint.

The mouse, or the entity traversing the maze, can only move in two directions: forward (downward in the matrix) or to the right (forward in the matrix). When the mouse encounters a cell with two possible directions to choose from, it will continue moving in one direction and create a new thread with another color or branch to explore the second possible direction. This branching mechanism allows the mouse to explore multiple paths simultaneously.

To optimize the execution of the maze-solving algorithm, the number of threads or branches created should be limited based on the number of available processors. This ensures that the program efficiently utilizes the computational resources by restricting the parallel execution of threads or branches to a manageable number.

Overall, the problem involves modeling a maze as a matrix, defining the start and end points with yellow color and determining the movement

rules for the mouse and you can show it with change in colors, implementing a branching mechanism for multiple paths, and optimizing the number of threads or branches based on the available processors.

Pseudocode:

==========

- Start Program

- Take maze size from user using keyboard

- create maze[n-1][n-1]

- visited_grids[n*2][2]

- final static end_point = [n-1,n-1]

- Take position of walls => [equation (row number = position / maze_size) (column_number = position % maze_size)]

- ArrayList Threads = new ArrayList

- Create Thread

- color = choose color

- Start running rat function([0,0], visited_grids, color)

- Start Thread

- Threads.add(Thread)

==========

runningRat (array int position, array visited_grids, color) {

- current_visited_grids = visited_grids.clone

- visit and color current position grid

if position != end_point {

- x = position[0]

- y = position[1]

- next = Null

- below = Null

- if (x + 1 <= n-1 && maze[x+1][y] != -1){

    below = [x+1, y]

}

- if (j + 1 <= n-1 && maze[x][y+1] != -1){

    next = [x, y+1]

}

- current_visited_grids = visited_grids

If next {

    If below {

        - Create Thread

        - new_color = choose color

        - running_rat(below, current_visited_grids, new_color)

        - Start Thread

        - Threads.add(Thread)

    }

    - running_rat(next, current_visited_grids)

}

Elif below {

    - Write position to visited_grids [critical section]

    - running_rat(below, current_visited_grids)

}

else {

- Threads.remove(Thread)

- Kill Thread

}

}

else{

- Threads.remove(Thread)

- Kill Thread

- visited_grids = current_visited_grids

}

}

-The algorithm being used is Breadth-First-Search (BFS).

-The complexiety is O(n).

-Here we use linear search.

-In this problem we don't find the optimal path, instead, we find the possible path for the goal and show it with green color(to get out of the maze) .

<u>Some Considerable Notes:</u>

The maximum number of threads accessing the same grid is 2.

If there's two possible directions the rat can cross, we will continue with the parent thread wether in the forward or below direction based on coulmn.

The child thread's path is based on row.

The following is a summary for what's inside the Grid Class and what some lines of code or methods do:

GUI :

In the first page

- You enter the size of maze
- Click on submit

In the second page

- Appears the maze with size That you enter it
- You can determine the location of wall by simply clicking on any square, and its color will change to gray and appear with a value of -1
- Click on start

What happens with the threads?

- The first thread will move, and if it meets two paths that are allowed to move, it will move in one direction and create a thread with another color to move in the other direction.

In the last page

- The maze will appear in colors and numbers
- The gray color represents the wall, the green color expresses the solution that goes from the starting point to the ending point, and the remaining colors each express the creation of a new thread.

We have a class called "Grid": // Inherit Button for GUI

It has the following methods:

1) makeWall

2) getValue

3) acquireGrid

4) releaseGrid

5) isValidGrid

6) visit

And here's what each method is responsible for:

1) makeWall:

It makes each wall in the maze equals to -1.

And changes background to GRAY Color with "-1" text

2) getValue:

It returns the value of maze.

3) acquireGrid:

It checks if we can get the lock or not.

4) releaseGrid:

Unlock on the elements.

5) isValidGrid

It checks if the value of grid is equals to zero.

6) visit:

It checks if, in the first place, we can acquire/get the lock or not.

It also checks if the grid is valid or not.

If it is valid, put the value of id as text and choose for it color

---

Presenting what each method in Maze.java is responsible for:

private void ratInMaze(int[] position, int[][] visited_grids, int row, Color c)

-It takes 4 parameters:

1) an array of type integer called "position": -It indicates where the rat is.

2)a 2D array of type int called "visited_grids": -It represents the goal, where the rat should reach so it be out of maze.

3)a variable of type int called "row" : -It represents the available places for the rat.

4)an object of type Color called "c": -It colors the threads according to each ThreadID.

```
if (nextRow <= size-1 && this.buttons[nextRow][y].isValidGrid()){
    // it checks if this is the last row and also if it is valid or not.
                belowCondition = true;
    // if true it makes the belowCondition true.
                below = new int[]{nextRow, y};
     //it makes the (below) equal the next move that the rat has reached.
            }
```

```
if (nextColumn <= size-1 && this.buttons[x][nextColumn].isValidGrid()){
     // it checks if this is the last column and also if it is valid or not.
                nextCondition = true;
    // if true it makes the nextCondition true.
                next = new int[]{x, nextColumn};
  //it makes the (next) equal the next move that the rat has reached.
```

```
if (nextCondition){
                    if (belowCondition){
```

//if the nextCondition & belowCondition is true, it creates a new thread cause we have two possible directions. the parent thread will not stop, it will contimue in either of the two paths, and the new thread or child thread will move in the other path.

```
            }
                    }
```

```
this.ratInMaze(next, current_visited_grids, row, c);
            //parent thread
```

```
else if (belowCondition){
                    this.ratInMaze(below, current_visited_grids, row, c);
                     //child thread
            }
```

```
else{
                    Thread.currentThread().interrupt();
        // if both of the above conditions which are "nextCondition" and
        // "nextCondition" are false, it will kill the thread cause it will be a deadend.
            }
```

```
else{
            this.condition = true;
            this.pathValues = current_visited_grids.clone();
            // if there's a valid path for the goal(to get out of mzae), therefore it will
            //print this mentioned path.
            Thread.currentThread().interrupt(); // kill all the threads
```

```
synchronized(this.threads){
            this.threads.remove(Thread.currentThread());
          // remove color of all created threads
        }
```