CSIS 3103

Ch 5:
Applications of Recursion

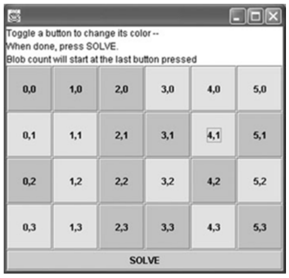
Counting Cells in a Blob

- Process an image presented as a two-dimensional array of color values
- Information in the image may come from
  - an X-ray
  - an MRI
  - satellite imagery
  - etc.
- The goal is to determine the size of any area in the image that is considered abnormal because of its color values

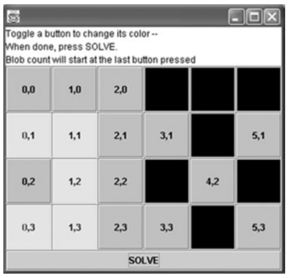
Counting Cells in a Blob

- Each cell in a two-dimensional grid contains either a normal background color or a second abnormal color
- A *blob* is a collection of contiguous abnormal cells
- A user will enter the x, y coordinates of a cell in the blob, and the program will determine the count of all cells in that blob

Counting Cells in a Blob


Toggle a button to change its color --
When done, press SOLVE.
Blob count will start at the last button pressed

0,0	1,0	2,0	3,0	4,0	5,0
0,1	1,1	2,1	3,1	4,1	5,1
0,2	1,2	2,2	3,2	4,2	5,2
0,3	1,3	2,3	3,3	4,3	5,3


Toggle a button to change its color --
When done, press SOLVE.
Blob count will start at the last button pressed

0,0	1,0	2,0	■	■	■
0,1	1,1	2,1	3,1	■	5,1
0,2	1,2	2,2	■	4,2	■
0,3	1,3	2,3	3,3	■	5,3

Counting Cells in a Blob - Design

Algorithm for `countCells(x, y)`

```
if cell at (x, y) is outside the grid
      the result is 0
else if color of cell at (x, y) ≠ abnormal color
      the result is 0
else
      set color of cell at (x, y) to temporary color
      the result is 1 + number of cells in each piece of
      the blob that includes a nearest neighbor
```

Counting Cells in a Blob - Implementation

```
    /** Finds the number of cells in the blob at (x,y).
        pre: Abnormal cells are in ABNORMAL color;
             Other cells are in BACKGROUND color.
        post: All cells in the blob are in the TEMPORARY color.
        @param x The x-coordinate of a blob cell
        @param y The y-coordinate of a blob cell
        @return The number of cells in the blob that contains (x, y)
    */
    public int countCells(int x, int y) {
        int result;

        if (x < 0 || x >= grid.getNCols()
                || y < 0 || y >= grid.getNRows())
            return 0;
        else if (!grid.getColor(x, y).equals(ABNORMAL))
            return 0;
        else {
            grid.recolor(x, y, TEMPORARY);
            return 1
                + countCells(x - 1, y + 1) + countCells(x, y + 1)
                + countCells(x + 1, y + 1) + countCells(x - 1, y)
                + countCells(x + 1, y) + countCells(x - 1, y - 1)
                + countCells(x, y - 1) + countCells(x + 1, y - 1);
        }
    }
}
```

Backtracking

- A systematic trial and error search for a solution
- Finding a path through a maze
  - To walk through a maze, you will probably walk down a path as far as you can go
  - Eventually, you will reach your destination or you won't be able to go any farther
  - If you can't go any farther, you will need to consider alternative paths
- Backtracking is a systematic, nonrepetitive approach to trying alternative paths and eliminating them if they don't work

Backtracking (cont.)

- If you never try the same path more than once, you will eventually find a solution path if one exists
- Recursion provides a relatively straightforward implementation of backtracking
- Each activation frame is used to remember the choice that was made at that particular decision point

Finding a Path through a Maze

- Problem
  - Use backtracking to find a display the path through a maze
  - From each point in a maze, you can move to the next cell in a horizontal or vertical direction, if the cell is not blocked

Finding a Path through a Maze (cont.)

- Analysis
  - The maze will consist of a grid of colored cells
  - The starting point is at the top left corner (0,0)
  - The exit point is at the bottom right corner `(getNCols() – 1, getNRow -1)`
  - All cells on the path will be `BACKGROUND` color
  - All cells that represent barriers will be `ABNORMAL` color
  - Cells that we have visited will be `TEMPORARY` color
  - If we find a path, all cells on the path will be set to `PATH` color

Recursive Algorithm for `findMazePath(x, y)`

```
if current cell is outside the maze
    return false (you are out of bounds)
else if  current cell is part of the barrier or has already been visited
    return false (you are off the path or in a cycle)
else if  current cell is the maze exit
    recolor it to the path color and return
else //  Try to find a path from the current cell to the exit:
    mark current cell by recoloring it to the path color
    for each neighbor of the current cell
        if a path exists from the neighbor to the maze exit
            return true
    // No neighbor of the current cell is on the path
    recolor current cell to the temporary color (visited) and
    return false
```