Maze Generation and Solving

Jackson Gregory
gregoryj17@asmsa.org

Objectives

- Create a program that can both generate and solve mazes. The maze were created with black pixels representing walls and white pixels representing paths. The solver outputted an image with blue pixels representing the solution and red pixels representing visited dead ends.

Specifications

- A maze was defined to be a path such that each point had exactly one direct path to each other point in the maze.
- The maze size was to be a height of the nth Fibonacci number nodes and a width of the (n+1)th Fibonacci number nodes. The picture itself, including walls, was to be 2*width+1 pixels by 2*height+1 pixels. The entrance was to be attached to the top-left node, pointing left, while the exit was attached to the bottom-right node, pointing right.

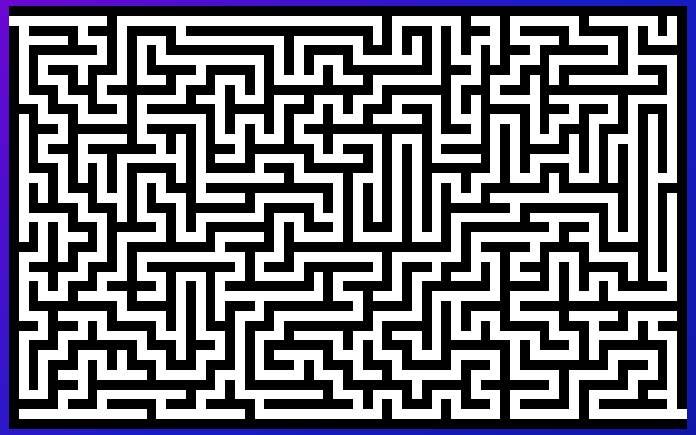
Generation

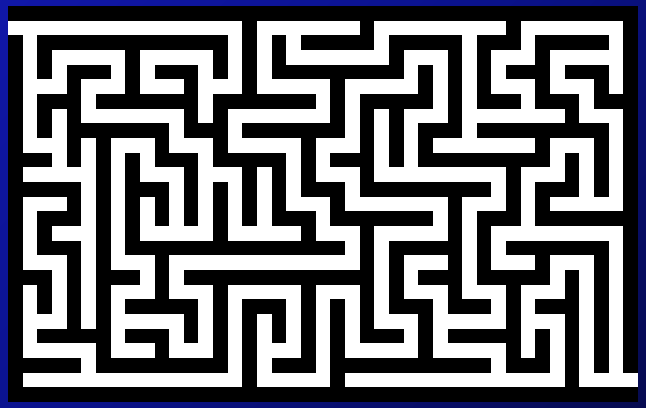
- Similar to Recursive Backtracking
- Random starting point
- For each point:
  - Point is added to the stack
  - If it can be branched from, a path is made, and the new point is added to the stack
  - If the point is a dead end, it is removed from the stack and the new last point in the stack is used

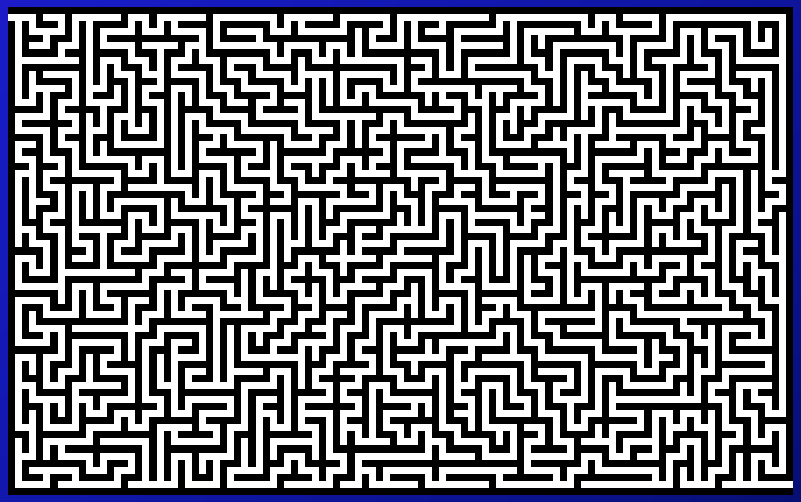
Generation Code Snippet

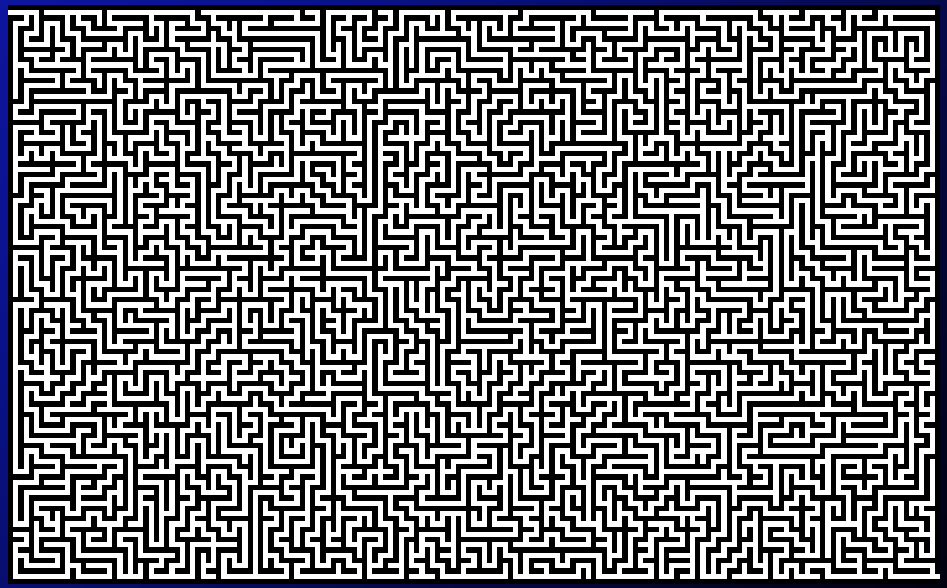
```
public static void drawMaze(int row, int col) {
    maze[row][col] = 255;
    ArrayList<Integer> stack = new ArrayList<Integer>();
    stack.add(row*mw+col);
    while (!stack.isEmpty()) {
        int coords = stack.get(stack.size() - 1);
        ArrayList<Integer> possibles = getPossibles(coords/mw, coords%mw);
        if (!possibles.isEmpty()) {
            int ncoords = possibles.get((int) (Math.random() * possibles.size()));
            maze[(ncoords/mw+coords/mw)/2][(ncoords%mw+coords%mw)/2]=255;
            maze[ncoords/mw][ncoords%mw] = 255;
            stack.add(ncoords);
        } else {
            stack.remove(stack.size() - 1);
        }
    }
    maze[1][0] = 255;
    maze[2 * h - 1][2 * w] = 255;
}
```

Sample Mazes









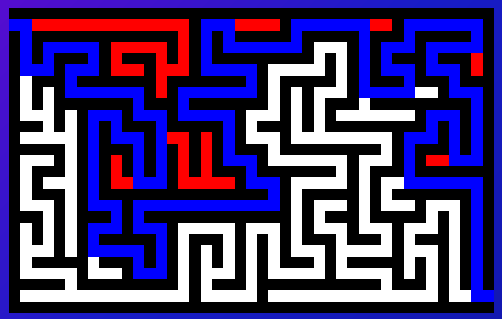
Solving

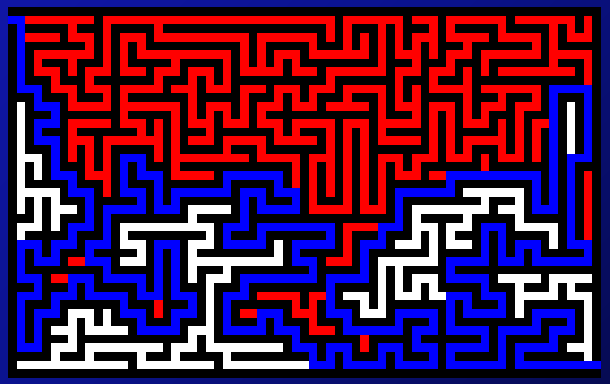
- “Left-Hand Rule”
- Solver keeps track of current location and last location
- Checks clockwise for an unvisited location and moves the first one it finds
- If all are visited, takes the first spot it finds that isn’t already marked as a dead end
- Marks visited spots (blue) and known dead ends (red)

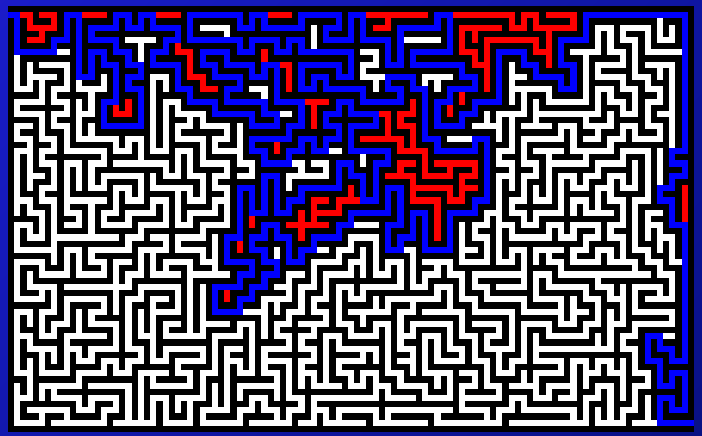
Solving Code Snippet

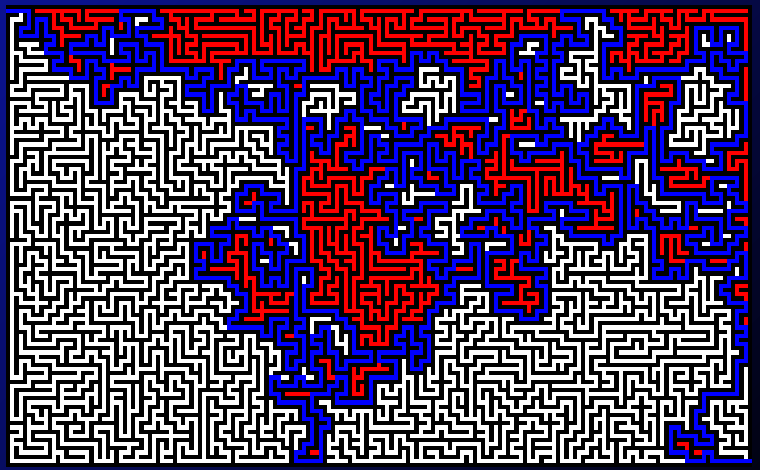
```
public static void solveMaze(){
    int row=1;
    int col=0;
    int dir=getDir(row, col);
    boolean dead=false;
    while(!isFinished()){
        dead=isDead(row, col);
        mark(row, col, dead);
        row+=dir==0?-1:dir==2?1:0;
        col+=dir==3?-1:dir==1?1:0;
        lastDir=(dir+2)%4;
        dir=getDir(row,col);
    }
    solved=true;
}
```

Sample Solves









Results

Maze Number	Width (pixels)	Height (pixels)	Solve Time (seconds)
6	43	27	0.011708654
7	69	43	0.020698688
8	111	69	0.061403864
9	179	111	0.108155803
10	289	179	0.304525350
11	467	289	0.601835068
12	755	467	1.295408737
13	1221	755	2.590986817
14	1975	1221	7.444378621
15	3195	1975	16.641055495
16	5169	3195	40.683680468
17	8363	5169	109.466671532
18	13531	8363	311.939386977

Insights

- Initially, the program would sometimes skip the exit in favor of exploring a dead end to its left. This was changed so that when the solver finds itself next to the exit, the exit will take priority over the left-hand rule.
- The program could be further optimized by replacing the left-hand rule with a pathfinding algorithm such as A*

Generation/Solving Videos

- This youtube playlist contains animations of the generation and solving algorithms at work.

Maze Generation and Solving

Jackson Gregory
gregoryj17@asmsa.org