Code

```java
package final_for_223J;

public class Main {

    public static void main(String[] args) {
   // write your code here
        new PathFinder();
    }
}
```

```java
package final_for_223J;

import java.awt.*;

public enum JNodeState {
    EMPTY(Color.white), WALL(Color.blue),
    START(Color.green), END(Color.red),
    VISITED(Color.yellow), ERROR(new Color(204, 0, 0));

    private Color color;
    JNodeState(Color _color)
    {
        color = _color;
    }

    public Color getColor(){ return color;}
}
```

```java
package final_for_223J;

import javax.swing.*;
import java.util.*



public class JNode extends JButton {
    private int row, col;
    private JNodeState state;
    private JNode previous;
    private int gCost, hCost, fCost;

    public JNode(int _row, int _col, JNodeState _state) {
        super();
        setBackground(_state.getColor());

        row = _row;
        col = _col;
        state = _state;

        previous = null;
        gCost = 0; //Distance to start node
        hCost = 0; //Distance to end node
        fCost = 0; //gCost + hCost
    }


    public JNode(int _row, int _col){
        this(_row, _col, JNodeState.EMPTY);
    }

    public void setState(JNodeState _state){
        if (state == _state) return;
        state = _state;
```

```
        setBackground(state.getColor());
    }

    public void setPrevious(JNode _previous){ previous = _previous;}

    public void setG(int g)
    {
        gCost = g;
    }

    public void setH(int h)
    {
        hCost = h;
    }

    public void setF()
    {
        fCost = gCost + hCost;
    }

    public JNodeState getState(){ return state; }
    public JNode getPrevious(){ return previous;}
    public int getRow(){ return row;}
    public int getCol(){ return col; }
    public int getG(){ return gCost; }
    public int getH(){ return hCost; }
    public int getF(){ return fCost; }

    public String toString()
    {
        return "(" + row + ", " + col + ")";
    }

    public boolean equals(Object o)
    {
        if (o == this)
            return true;

        if (!(o instanceof JNode))
            return false;

        JNode c = (JNode)o;

        boolean sameRow = row == c.getRow();
        boolean sameCol = col == c.getCol();
        return sameRow && sameCol;
    }
}
```

```java
package final_for_223J;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.ArrayList;
import java.util.Collections;

public class PathFinder extends JFrame implements MouseListener, ItemListener {

    //Size of the frame
    private final int FRAME_WIDTH = 500;
    private final int FRAME_HEIGHT = 650;

    //All the panels in the frame
    private JPanel instructionsPanel = new JPanel();
    private JPanel configPanel = new JPanel();
    private JPanel gridSizesPanel = new JPanel();
    private JPanel drawDelayPanel = new JPanel();
    private JPanel boxPanel = new JPanel();
    private JPanel gridPanel = new JPanel();

    //Grid information i.e. size & gap of the grid
    private int gridSize = 10;
    private int gridGap = 1;
    private JNode [] [] grid = new JNode[gridSize][gridSize];
    private ArrayList<JNode> path = new ArrayList<JNode>(gridSize * gridSize);

    //Main nodes for the path to find
    private JNode startNode = null;
    private JNode endNode = null;

    //Shows that we found an error during the A* algorithm
    private Boolean foundError = false;

    //Variables that are associated with drawing the path
    private Timer tracePath, followTrail;
    private long startSearch, finishSearch;
    private int currentIndex = 0;
    private int delay = 100;

    //Instructions that tell the user what to do and what is going on
    private JLabel rightInstruction = new JLabel("Right Click to Add Wall ", JLabel.CENTER);
    private JLabel leftInstruction = new JLabel(" Left Click to Add Start ", JLabel.CENTER);

    //Swing components with the grid size panel
    private JLabel gridSizesLabel = new JLabel("Grid Size", JLabel.CENTER);
    private JSlider gridSizesSlider = new JSlider(5, 50, gridSize);

    //Swing components for the draw delay panel
    private JLabel drawDelayLabel = new JLabel("Delay (in milliseconds)", JLabel.CENTER);
    private JSlider drawDelaySlider = new JSlider(0, 500, delay);

    //Checkboxes that go below the draw delay panel and above the instructions
    private JCheckBox includeDiagonals = new JCheckBox("Include Diagonals");
    private JCheckBox generateMaze = new JCheckBox("Generate Maze");


    public PathFinder() {
        super("Jalen's Pathfinder");
        setSize(FRAME_WIDTH, FRAME_HEIGHT);
        setResizable(false);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);

        setLayout(new BorderLayout());
        configPanel.setLayout(new BorderLayout(0, 20));
        gridSizesPanel.setLayout(new BorderLayout(0, 1));
        drawDelayPanel.setLayout(new BorderLayout(0, 1));
        boxPanel.setLayout(new BorderLayout(10, 0));
        instructionsPanel.setLayout(new BorderLayout(0, 10));
```

```
gridPanel.setLayout(new GridLayout(gridSize, gridSize, gridGap, gridGap));

leftInstruction.setFont(new Font("Microsoft Sans Serif", Font.BOLD, 15));
rightInstruction.setFont(new Font("Microsoft Sans Serif", Font.BOLD, 15));
gridSizesLabel.setFont(new Font("Microsoft Sans Serif", Font.PLAIN, 14));
gridSizesSlider.setFont(new Font("Microsoft Sans Serif", Font.PLAIN, 12));
drawDelayLabel.setFont(new Font("Microsoft Sans Serif", Font.PLAIN, 14));
drawDelaySlider.setFont(new Font("Microsoft Sans Serif", Font.PLAIN, 12));
includeDiagonals.setFont(new Font("Microsoft Sans Serif", Font.PLAIN, 14));
generateMaze.setFont(new Font("Microsoft Sans Serif", Font.PLAIN, 14));

instructionsPanel.add(configPanel, BorderLayout.NORTH);
instructionsPanel.add(leftInstruction, BorderLayout.WEST);
instructionsPanel.add(rightInstruction, BorderLayout.EAST);

configPanel.add(gridSizesPanel, BorderLayout.NORTH);
gridSizesPanel.add(new JPanel(), BorderLayout.NORTH);
gridSizesPanel.add(gridSizesLabel, BorderLayout.CENTER);
gridSizesPanel.add(gridSizesSlider, BorderLayout.SOUTH);
gridSizesSlider.setMajorTickSpacing(5);
gridSizesSlider.setMinorTickSpacing(1);
gridSizesSlider.setPaintTicks(true);
gridSizesSlider.setSnapToTicks(true);
gridSizesSlider.setPaintLabels(true);
gridSizesSlider.addMouseListener(this);

configPanel.add(drawDelayPanel, BorderLayout.CENTER);
drawDelayPanel.add(drawDelayLabel, BorderLayout.NORTH);
drawDelayPanel.add(drawDelaySlider, BorderLayout.SOUTH);
drawDelaySlider.setMajorTickSpacing(50);
drawDelaySlider.setMinorTickSpacing(10);
drawDelaySlider.setPaintTicks(true);
drawDelaySlider.setPaintLabels(true);
drawDelaySlider.addMouseListener(this);

configPanel.add(boxPanel, BorderLayout.SOUTH);
boxPanel.add(includeDiagonals, BorderLayout.WEST);
boxPanel.add(generateMaze, BorderLayout.EAST);
includeDiagonals.setHorizontalAlignment(JCheckBox.LEFT);
generateMaze.setHorizontalAlignment(JCheckBox.LEFT);
generateMaze.addItemListener(this);

add(instructionsPanel, BorderLayout.NORTH);
add(gridPanel, BorderLayout.CENTER);
for (int i = 0; i < gridSize; i++) {
    for (int j = 0; j < gridSize; j++) {
        grid[i][j] = new JNode(i, j);
        grid[i][j].addMouseListener(this);
        gridPanel.add(grid[i][j]);
    }
}

//Animation where it highlights each node in the path frame by frame
tracePath = new Timer(delay, new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        if (currentIndex < path.size()) {
            path.get(currentIndex++).setState(JNodeState.VISITED);
            gridPanel.revalidate();
            gridPanel.repaint();
        } else {
            currentIndex = 0;
            rightInstruction.setText("Following Trail... ");

            followTrail.start();
            tracePath.stop();
        }
    }
});
```

```
    /*Animation where it removes all of the highlights in order frame by frame
      Then, it makes the last node the new start node
    */
    followTrail = new Timer(delay, new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            if (currentIndex < path.size() - 1) {
                path.get(currentIndex++).setState(JNodeState.EMPTY);
                gridPanel.revalidate();
                gridPanel.repaint();
            } else {
                path.get(currentIndex).setState(JNodeState.START);
                startNode = path.get(currentIndex);
                endNode = null;

                gridPanel.revalidate();
                gridPanel.repaint();

                currentIndex = 0;
                path.clear();

                leftInstruction.setText(" Left Click to Add End ");
                rightInstruction.setText("Right Click to Add Wall ");
                gridSizesSlider.setEnabled(true);
                generateMaze.setEnabled(true);

                followTrail.stop();
            }
        }
    });

    setVisible(true);
}

//For more details on the Recursive Algorithm, visit...
//http://weblog.jamisbuck.org/2011/1/12/maze-generation-recursive-division-algorithm
public void GenerateMaze()
{
    //1. Begin with a empty grid
    Divide(0, 0, gridSize - 1, gridSize - 1);
    gridPanel.revalidate();
    gridPanel.repaint();
}

public void Divide(int minRow, int minCol, int maxRow, int maxCol)
{
    int width = maxRow - minRow;  //Width of the wall
    int height = maxCol - minCol; //Height of the wall

    if  (width < 2 || height < 2)
    {
        //Maze has reached its desired resolution
        return;
    }

    //2. Split the maze into fourths with a horizontal wall and a vertical wall
    int wallRow = minRow + 1 + (int) (Math.random() * (maxRow - minRow - 1));
    int wallCol = minCol + 1 + (int) (Math.random() * (maxCol - minCol - 1));
    int row = minRow;
    int col = minCol;
    while (row <= maxRow) grid[row++][wallCol].setState(JNodeState.WALL);
    while (col <= maxCol) grid[wallRow][col++].setState(JNodeState.WALL);

    //3. Add two passages through each side of the horizontal and the vertical wall
    int northHole =  minRow + (int) (Math.random() * (wallRow - minRow));
    int southHole = (wallRow + 1) + (int) (Math.random() * (maxRow - wallRow - 1));
    int westHole  =  minCol + (int) (Math.random() * (wallCol - minCol));
    int eastHole  = (wallCol + 1) + (int) (Math.random() * (maxCol - wallCol - 1));
    grid[northHole][wallCol].setState(JNodeState.EMPTY);
    grid[southHole][wallCol].setState(JNodeState.EMPTY);
    grid[wallRow][westHole].setState(JNodeState.EMPTY);
```

```
        grid[wallRow][eastHole].setState(JNodeState.EMPTY);

        //4. Recursively divide each of the four sides until it reaches its desired resolution
        Divide(minRow, minCol, wallRow - 1, wallCol - 1);
        Divide(minRow, wallCol + 1, wallRow - 1, maxCol);
        Divide(wallRow + 1, minCol, maxRow, wallCol - 1);
        Divide(wallRow + 1, wallCol + 1, maxRow, maxCol);
    }

    //Gets the nodes nearby the current node. Includes diagonal nodes if requested
    public ArrayList<JNode> GetNeighborsOf(JNode current, Boolean includeDiagonals)
    {
        //Create a list that stores the nodes nearby the current node
        ArrayList<JNode> neighbors = new ArrayList<JNode>();

        //Get the row and column of the current node
        int row = current.getRow();
        int col = current.getCol();

        //Check if it is a valid move to go left, right, up, and down
        boolean left = col - 1 >= 0 && col - 1 < grid[row].length;
        boolean right = col + 1 >= 0 && col + 1 < grid[row].length;
        boolean up = row - 1 >= 0 && row - 1 < grid.length;
        boolean down = row + 1 >= 0 && row + 1 < grid.length;

        //If a direction is valid, add the node that goes to that direction in the neighbors list
        if(up)              neighbors.add(grid[row - 1][col]);
        if (down)           neighbors.add(grid[row + 1][col]);
        if (left)           neighbors.add(grid[row][col - 1]);
        if (right)          neighbors.add(grid[row][col + 1]);

        //If diagonals are included, add the NE, NW, SE, SW, nodes to the neighbors list
        if (includeDiagonals)
        {
            if (up && left) neighbors.add(grid[row - 1][col - 1]);
            if (up && right) neighbors.add(grid[row - 1][col + 1]);
            if (down && left) neighbors.add(grid[row + 1][col - 1]);
            if (down && right) neighbors.add(grid[row + 1][col + 1]);
        }

        //Return the neighbors
        return neighbors;
    }


    private int GetDistance(JNode start, JNode end) {
        if (start == null || end == null)
            return 0;

        int dstCol = Math.abs(start.getCol() - end.getCol()); //X
        int dstRow = Math.abs(start.getRow() - end.getRow()); //Y

        if (includeDiagonals.isSelected())
            return Math.max(dstRow, dstCol); //Diagonal Distance
        else
            return dstRow + dstCol; //Manhattan Distance
    }

    //Clears the nodes with the state specified in the grid
    public void ClearGrid(JNodeState state)
    {
        if (state == JNodeState.START) startNode = null;
        if (state == JNodeState.END) endNode = null;

        for (int i = 0; i < gridSize; i++) {
            for (int j = 0; j < gridSize; j++) {

                if (grid[i][j].getState() == state)
                    grid[i][j].setState(JNodeState.EMPTY);
            }
        }
```

```
        gridPanel.revalidate();
        gridPanel.repaint();
    }

    //Clears all nodes in the grid
    public void ClearGrid()
    {
        startNode = null;
        endNode = null;

        for (int i = 0; i < gridSize; i++) {
            for (int j = 0; j < gridSize; j++) {
                grid[i][j].setState(JNodeState.EMPTY);
            }
        }

        gridPanel.revalidate();
        gridPanel.repaint();
    }

    //A* Algorithm
    public void FindPath()
    {
        ArrayList<JNode> openList = new ArrayList<JNode>(); //Open = Nodes that need to be
evaluated
        ArrayList<JNode> closedList = new ArrayList<JNode>(); //Closed = Nodes that were already
evaluated

        //By default, the start node is evaluated first
        openList.add(startNode);

        //As long as they are nodes in the open list...
        startSearch = System.nanoTime();
        while (!openList.isEmpty())
        {
            //Find the node with the least f cost i.e. closest distance to the end node
            JNode currentNode = openList.get(0);
            for (int i = 0; i < openList.size(); i++)
            {
                if (openList.get(i).getF() < currentNode.getF())
                    currentNode = openList.get(i);
                else if(openList.get(i).getF() == currentNode.getF())
                {
                    if (openList.get(i).getG() > currentNode.getG())
                        currentNode = openList.get(i);
                    else if(openList.get(i).getG() == currentNode.getG())
                    {
                        if (openList.get(i).getH() < currentNode.getH())
                            currentNode = openList.get(i);
                    }
                }
            }

            //If the current node is the end node, then we have finally found the path.
            if (currentNode.equals(endNode))
            {
                //Now lets construct it so it can be traced
                ConstructPath();
                return;
            }

            //Otherwise, remove the current node out of the open list & add it to the closed list
            openList.remove(currentNode);
            closedList.add(currentNode);



            //Loop through each neighbor in the current node... (include diagonals if requested)
            for (JNode neighbor : GetNeighborsOf(currentNode, includeDiagonals.isSelected()))
            {
```

```
                //Skip this neighbor if its a wall or its already been evaluated
                if (neighbor.getState() == JNodeState.WALL || closedList.contains(neighbor))
                    continue;

                //Otherwise check if the new path in neighbor is shorter
                int possibleG = currentNode.getG() + GetDistance(neighbor, currentNode);

                //Make sure all neighbors that are not in the open list be added for evaluation
                if (!openList.contains(neighbor))
                    openList.add(neighbor);
                else
                {
                    //Skip this neighbor if it makes the path longer
                    if (possibleG >= neighbor.getG())
                        continue;
                }

                //G Cost = Distance from this node to the current node
                neighbor.setG(possibleG);

                //H Cost = Distance from this node to the end node
                neighbor.setH(GetDistance(neighbor, endNode));

                //F Cost = G Cost + H Cost
                neighbor.setF();

                //Set the neighbor's previous node to the current node
                neighbor.setPrevious(currentNode);
            }
        }

        //Path cannot be found since all nodes were evaluated except the end node
        startNode.setState(JNodeState.ERROR);
        endNode.setState(JNodeState.ERROR);

        startNode = null;
        endNode = null;

        foundError = true;
        leftInstruction.setText(" Cannot Find Path");

        //Repaint the grid to show our results in one click
        gridPanel.revalidate();
        gridPanel.repaint();
    }

    private void ConstructGrid(int size)
    {
        ClearGrid();

        gridPanel.removeAll();
        gridSize = size;
        gridPanel.setLayout(new GridLayout(gridSize, gridSize, gridGap, gridGap));
        grid = new JNode[gridSize][gridSize];

        for (int i = 0; i < gridSize; i++)
        {
            for (int j = 0; j < gridSize; j++)
            {
                grid[i][j] = new JNode(i, j);
                grid[i][j].addMouseListener(this);
                gridPanel.add(grid[i][j]);
            }
        }

        if (generateMaze.isSelected())
            GenerateMaze();

        gridPanel.revalidate();
        gridPanel.repaint();
    }
```

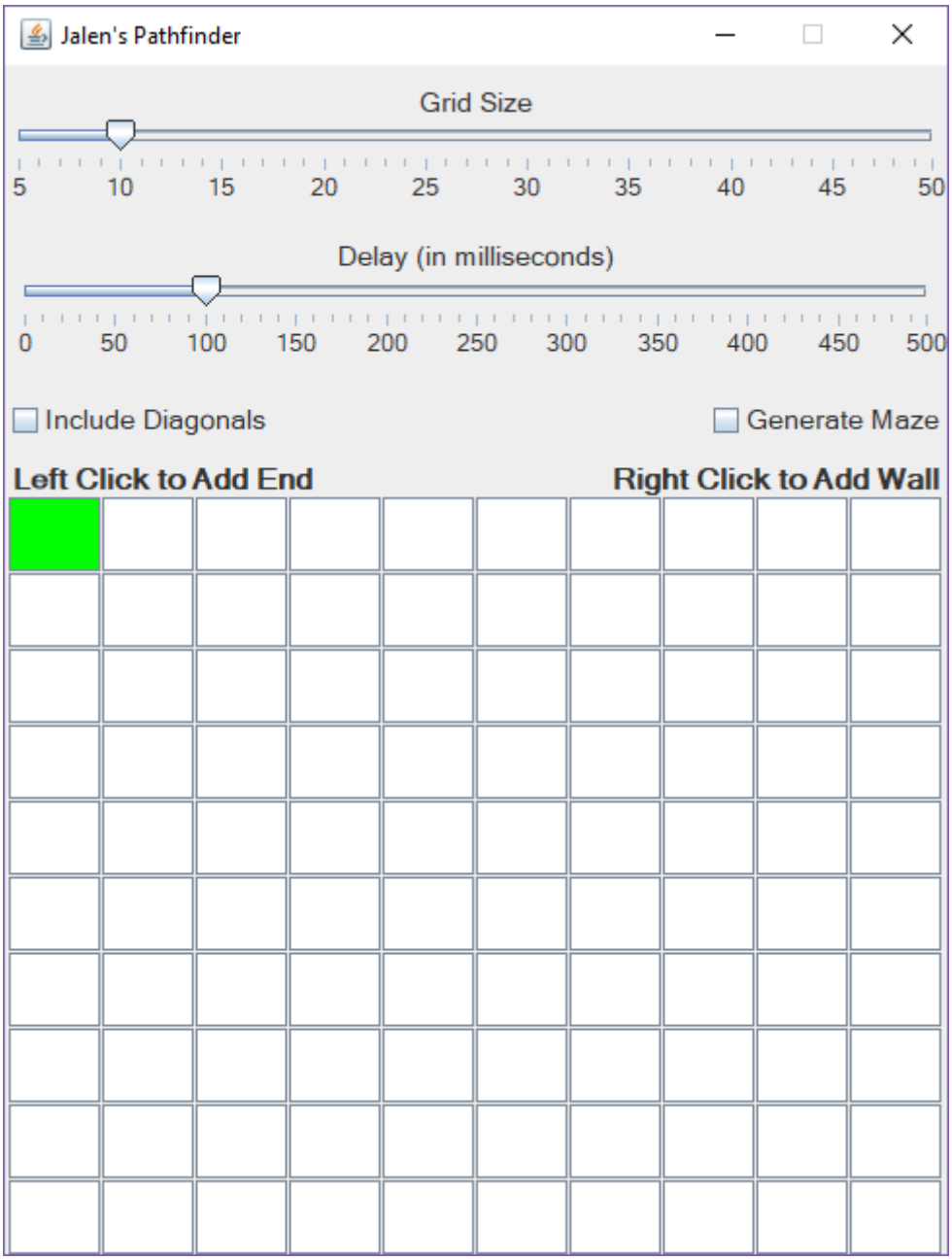
```
private void ConstructPath()
{
    //================================Updating the App=================================

    //Store the time it took to find the path in microseconds
    finishSearch = System.nanoTime();
    long searchElapsed = (finishSearch - startSearch) / (long)Math.pow(10, 3);

    //Update instructions to show the search time and tell the user the path is being traced
    leftInstruction.setText("Found Shortest Path in " + searchElapsed + " microseconds ");
    rightInstruction.setText("Retracing Path... ");

    //No maze generation or grid size changing allowed while path is being traced
    gridSizesSlider.setEnabled(false);
    generateMaze.setEnabled(false);

    //=========================Actual Construction of the Path=========================

    //Instantiate the current node then declare it as the end node
    JNode currentNode = endNode;

    //As long as the current node didn't reach to the start node...
    while (!currentNode.equals(startNode))
    {
        //Add the current to the path then traverse the current node to its parent
        path.add(currentNode);
        currentNode = currentNode.getPrevious();
    }

    //Reverse the path list since we traversed the current node backwards
    Collections.reverse(path);

    //Add the start node at the beginning of the path list
    path.add(0, currentNode);

    //============================Start Drawing the Path============================
    tracePath.start();
}

@Override
public void mousePressed(MouseEvent e) {
    if (tracePath.isRunning() || followTrail.isRunning())
        return;

    Object source = e.getSource();

    if (foundError)
    {
        ClearGrid(JNodeState.ERROR);
        foundError = false;
    }

    for (int i = 0; i < gridSize; i++)
    {
        for (int j = 0; j < gridSize; j++)
        {
            if (source == grid[i][j])
            {
                if (grid[i][j].getState() == JNodeState.EMPTY)
                {
                    if (SwingUtilities.isLeftMouseButton(e))
                    {
                        if (startNode == null)
                        {
                            startNode = grid[i][j];
                            startNode.setState(JNodeState.START);

                            leftInstruction.setText(" Left Click to Delete Start ");
                        }
                        else
```

```
                                {
                                    if (endNode == null) {
                                        endNode = grid[i][j];
                                        endNode.setState(JNodeState.END);

                                        FindPath();
                                    }
                                }
                            }
                            else if (SwingUtilities.isRightMouseButton(e))
                            {
                                if (grid[i][j].getState() == JNodeState.EMPTY)
                                    grid[i][j].setState(JNodeState.WALL);
                            }
                        }
                        else if(grid[i][j].getState() == JNodeState.START || grid[i][j].getState() ==
JNodeState.END)
                        {
                            if (grid[i][j].equals(startNode))
                            {
                                if (SwingUtilities.isLeftMouseButton(e))
                                {
                                    if (grid[i][j].getState() == JNodeState.START)
                                        ClearGrid(JNodeState.START);
                                    else
                                        ClearGrid(JNodeState.END);

                                    leftInstruction.setText(" Left Click to Add Start ");
                                }
                            }
                        }
                        else if(grid[i][j].getState() == JNodeState.WALL)
                        {
                            if (SwingUtilities.isRightMouseButton(e))
                                grid[i][j].setState(JNodeState.EMPTY);
                        }
                    }
                }
            }

            gridPanel.revalidate();
            gridPanel.repaint();
        }

        @Override
        public void mouseReleased(MouseEvent e) {
            Object source = e.getSource();

            if (source == gridSizesSlider)
            {
                int newSize = gridSizesSlider.getValue();
                ConstructGrid(newSize);
            }
            else if(source == drawDelaySlider)
            {
                delay = drawDelaySlider.getValue();
                tracePath.setDelay(delay);
                followTrail.setDelay(delay);
            }
        }

        @Override
        public void mouseEntered(MouseEvent e) {
            if (followTrail.isRunning() || tracePath.isRunning())
                return;

            Object source = e.getSource();
            for (int i = 0; i < gridSize; i++)
            {
                for (int j = 0; j < gridSize; j++)
                {
```

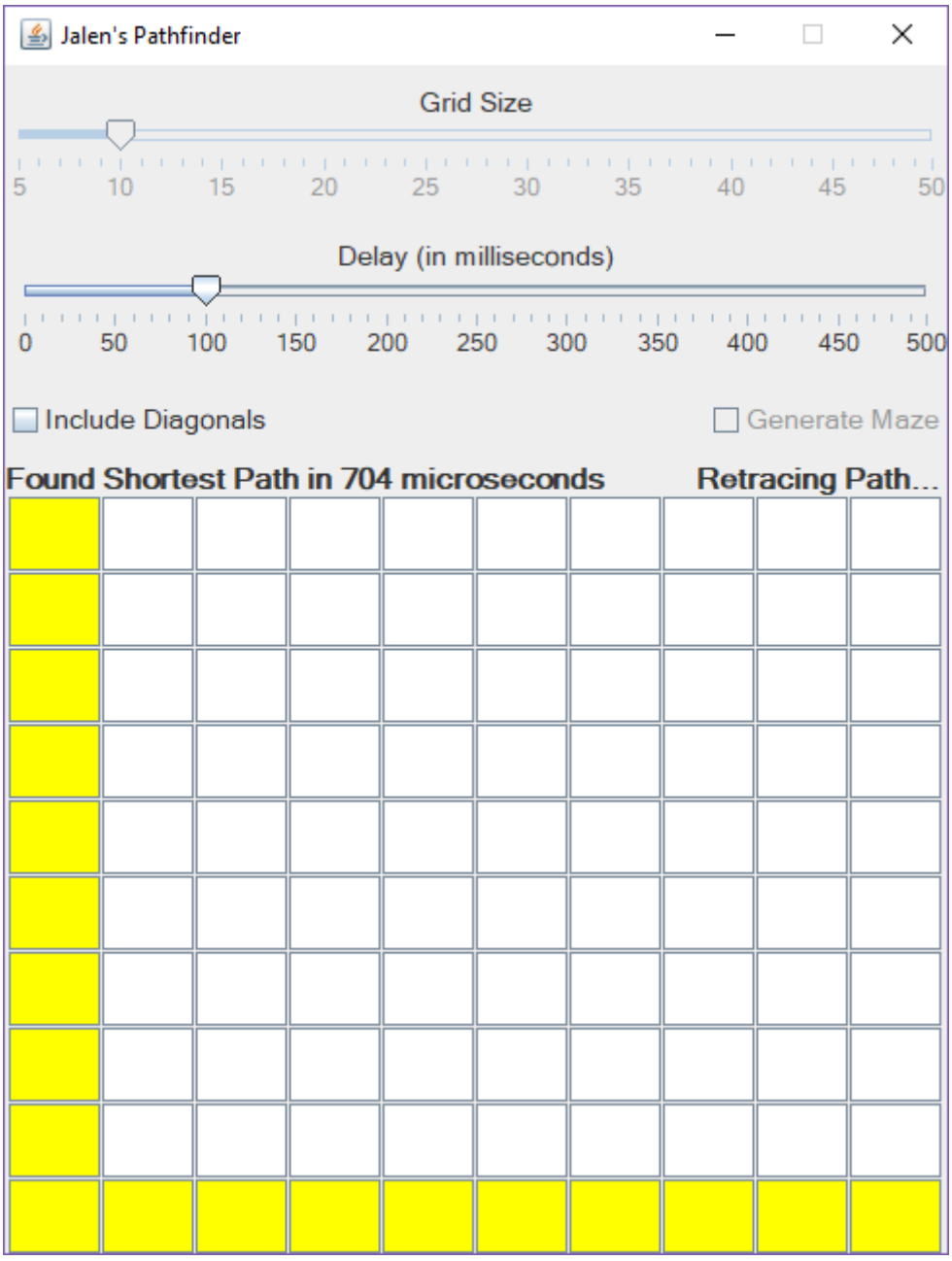
```
                if (source == grid[i][j])
                {
                    if (SwingUtilities.isRightMouseButton(e))
                    {
                        if (grid[i][j].getState() == JNodeState.EMPTY)
                        {
                            rightInstruction.setText("Right Drag To Add More Walls ");
                            grid[i][j].setState(JNodeState.WALL);
                        }
                        else if (grid[i][j].getState() == JNodeState.WALL)
                        {
                            rightInstruction.setText("Right Drag To Delete More Walls ");
                            grid[i][j].setState(JNodeState.EMPTY);
                        }
                    }
                    else
                    {
                        if (grid[i][j].getState() == JNodeState.EMPTY)
                            rightInstruction.setText("Right Click to Add Wall ");
                        else if (grid[i][j].getState() == JNodeState.WALL)
                            rightInstruction.setText("Right Click to Delete Wall ");
                    }

                    if (startNode != null && !followTrail.isRunning() && !tracePath.isRunning())
                    {
                        if (grid[i][j].equals(startNode))
                            leftInstruction.setText(" Left Click to Delete Start ");
                        else
                            leftInstruction.setText(" Left Click to Add End  ");
                    }
                }
            }
        }
    }
    @Override
    public void itemStateChanged(ItemEvent e) {
        if (e.getStateChange() == ItemEvent.SELECTED)
        {
            ClearGrid();
            GenerateMaze();
            gridPanel.revalidate();
            gridPanel.repaint();
        }
    }

    @Override
    public void mouseClicked(MouseEvent e) {

    }
    @Override
    public void mouseExited(MouseEvent e) {

    }

}
```

Sample Output

App Launched

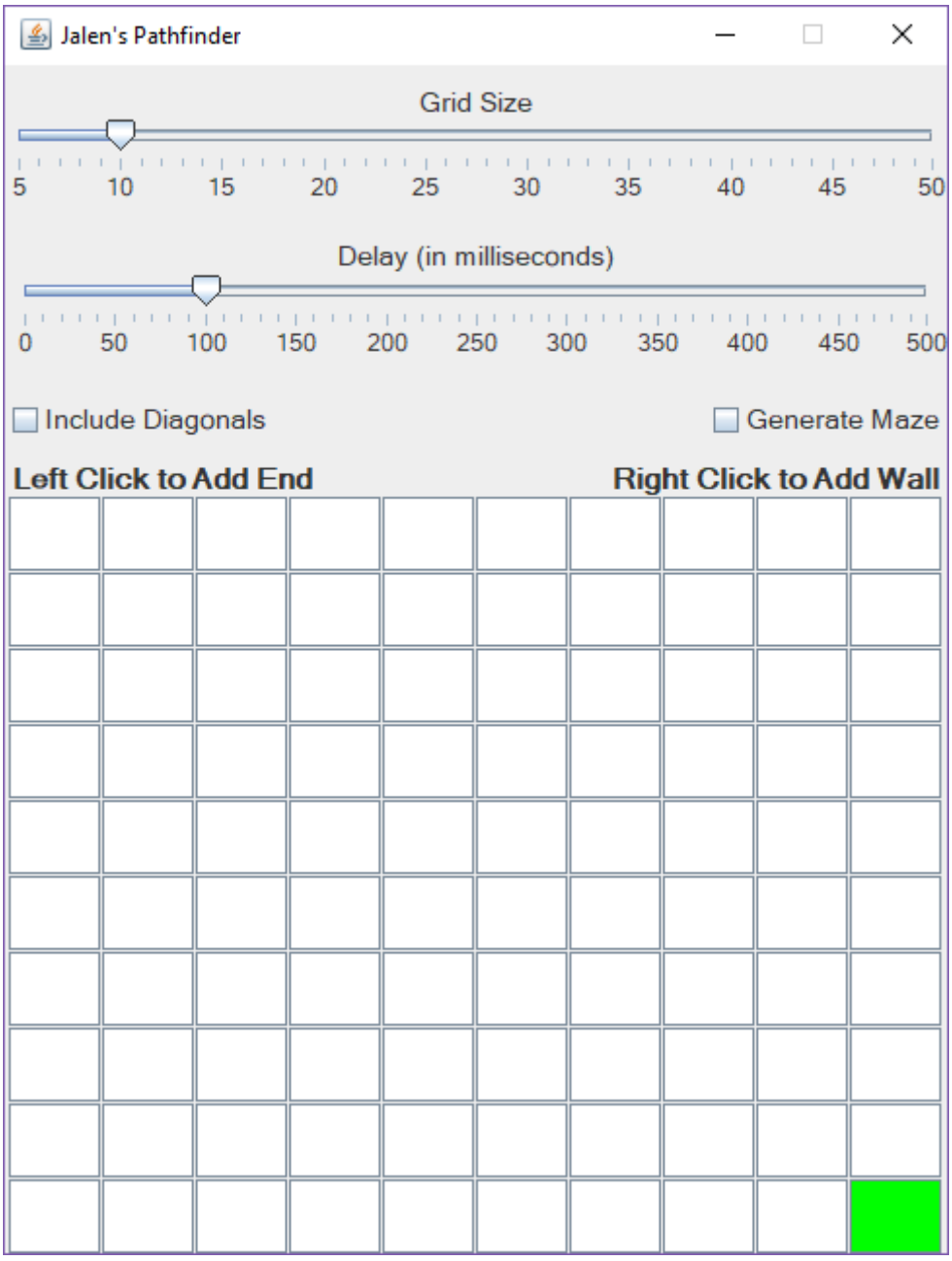


Added Start Node

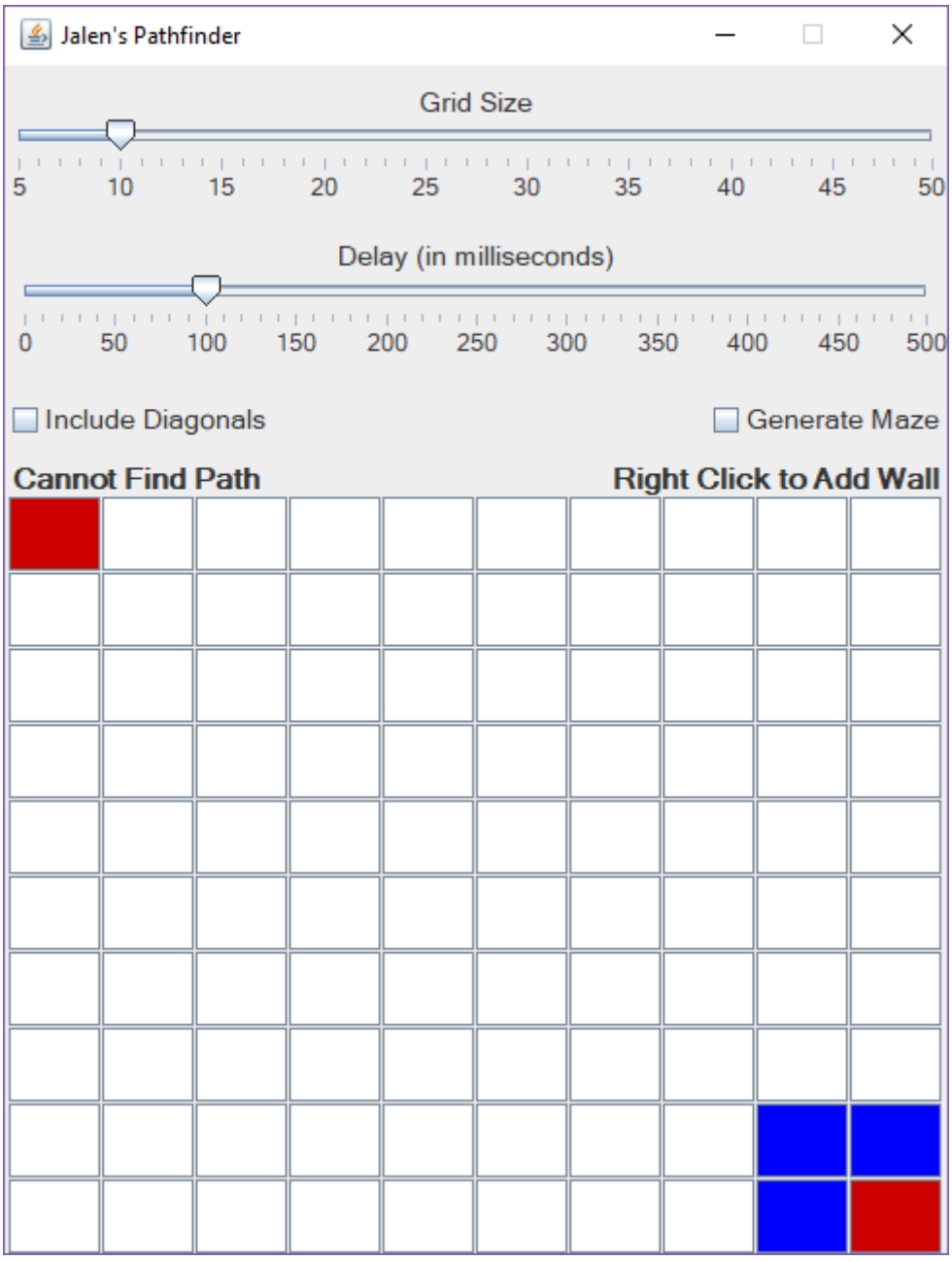


Added End Node

Found then Drawn Shortest Path



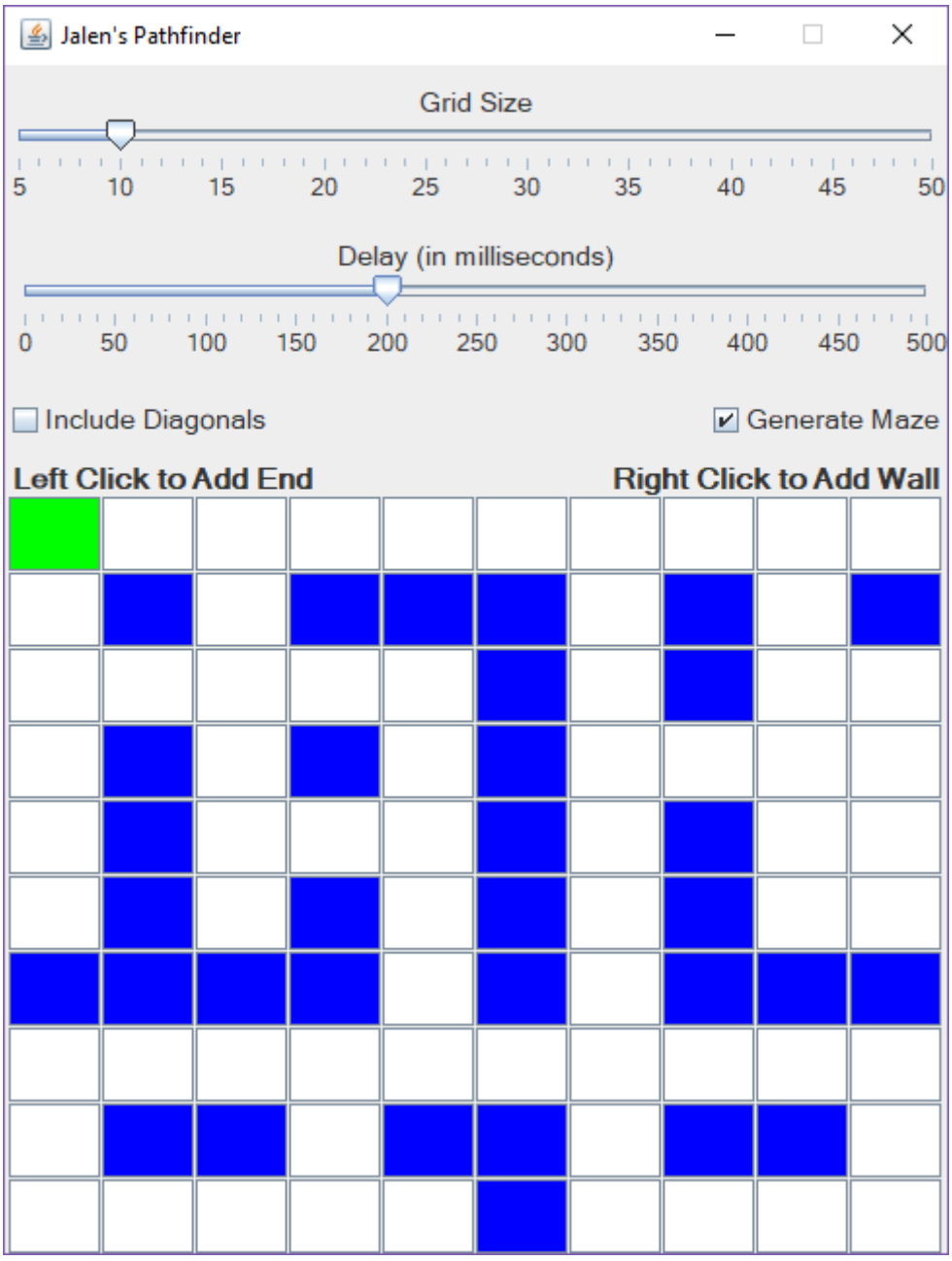
Start Node Moved to End Node's Spot



Added Start & End Node

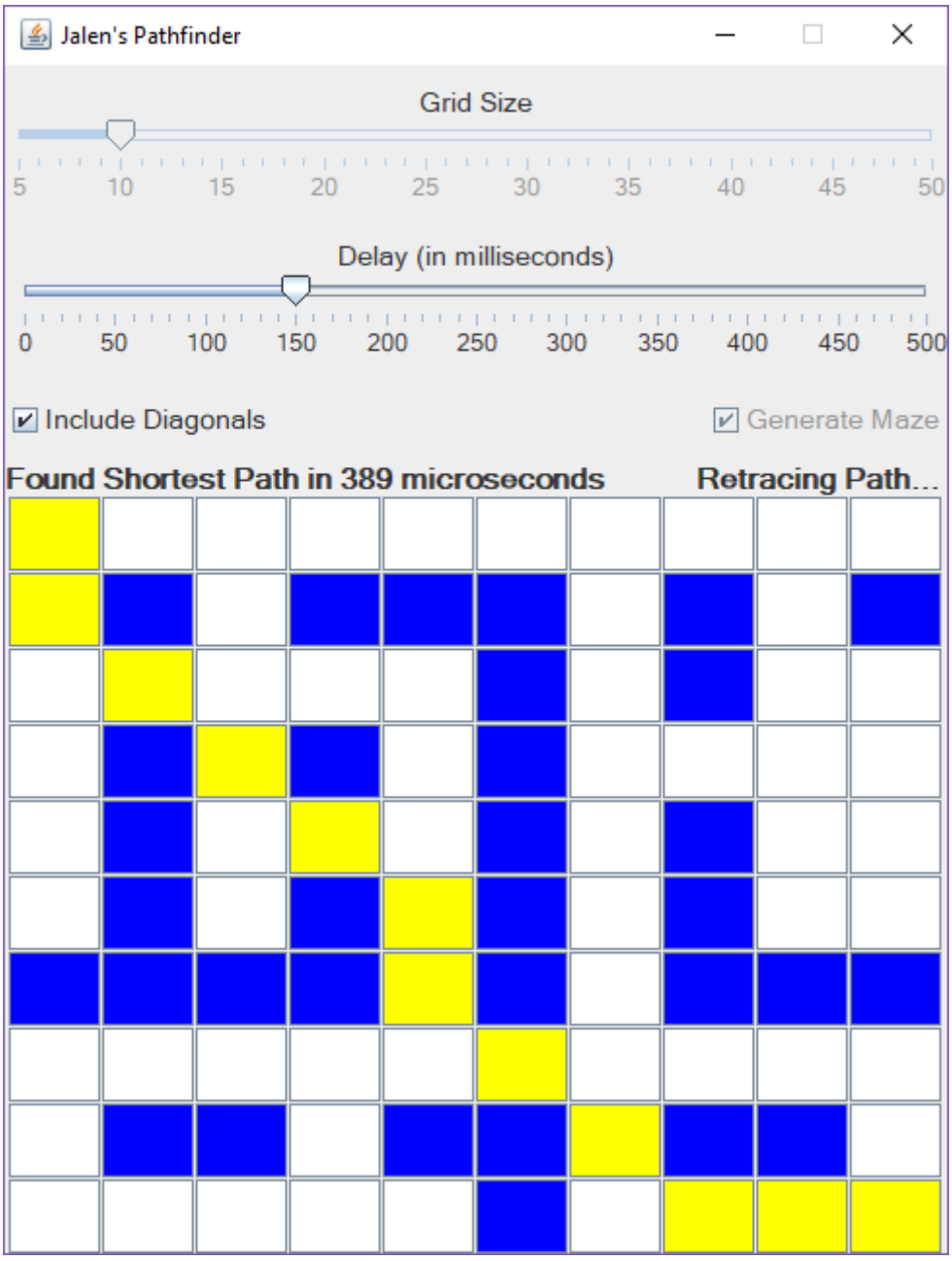
Added Walls Surrounding End Node

Couldn’t Find Path



Checked “Generate Maze” Box

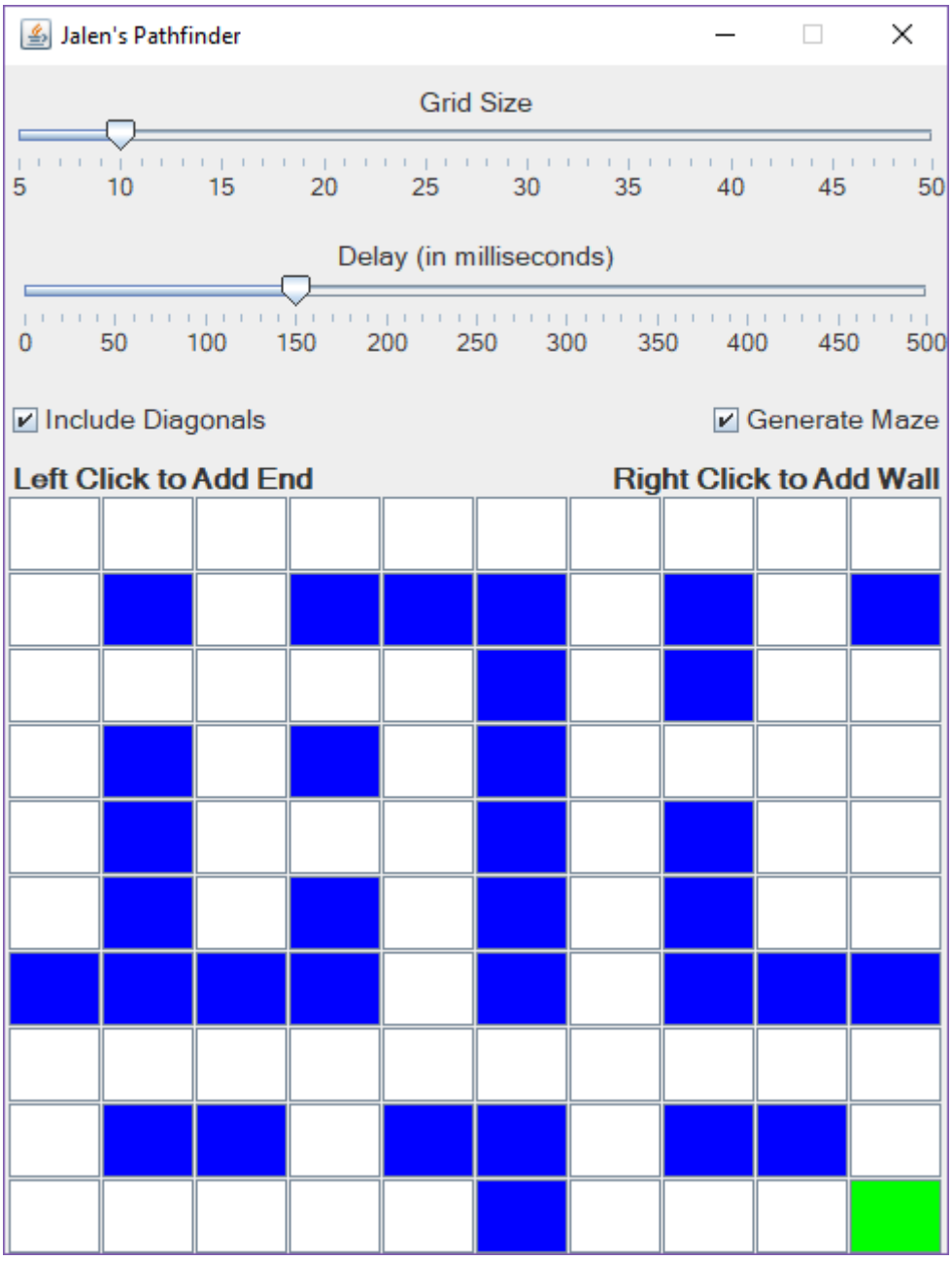
Added Start Node



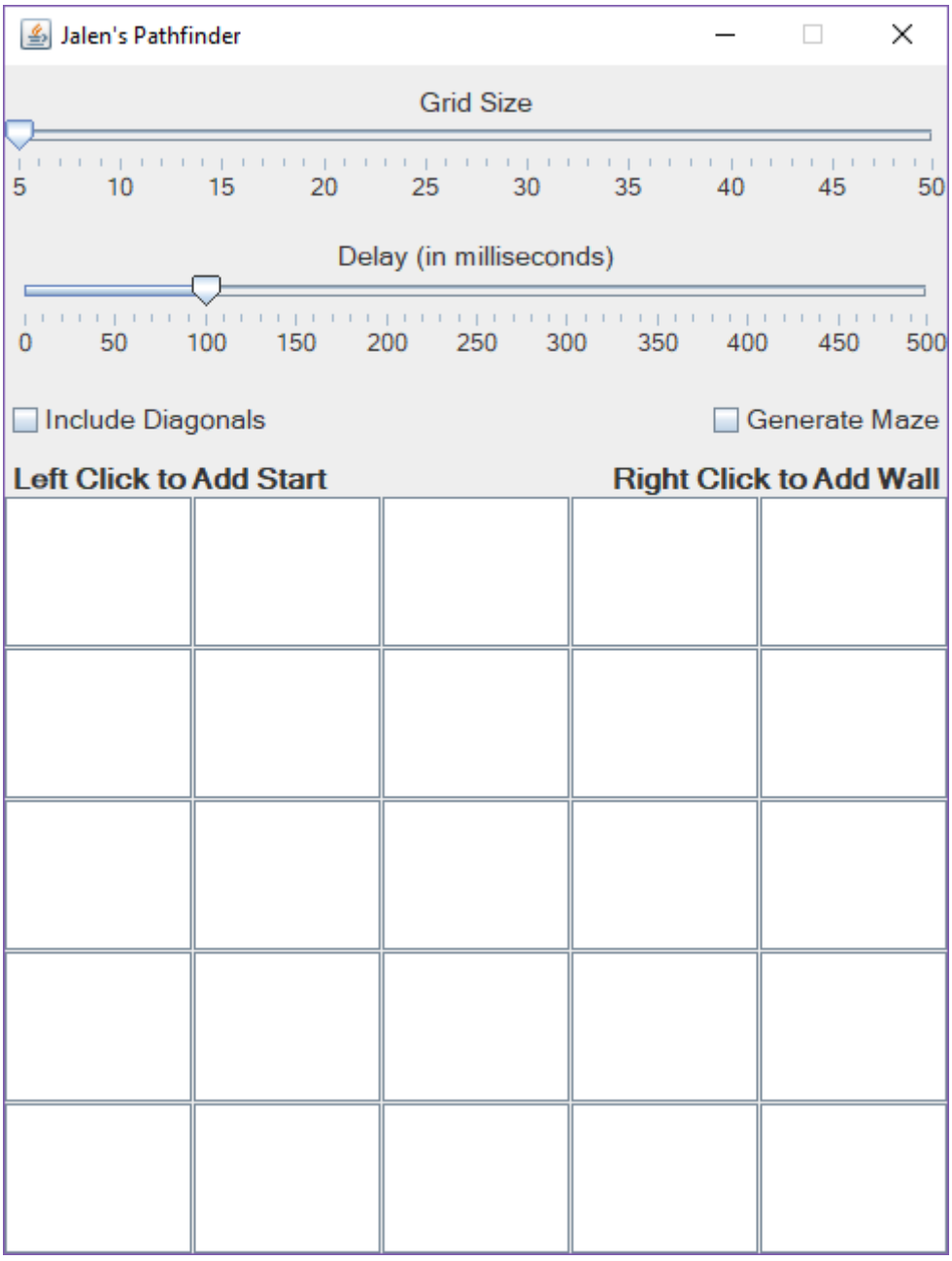
Checked “Included Diagonals” Box

Added End Node

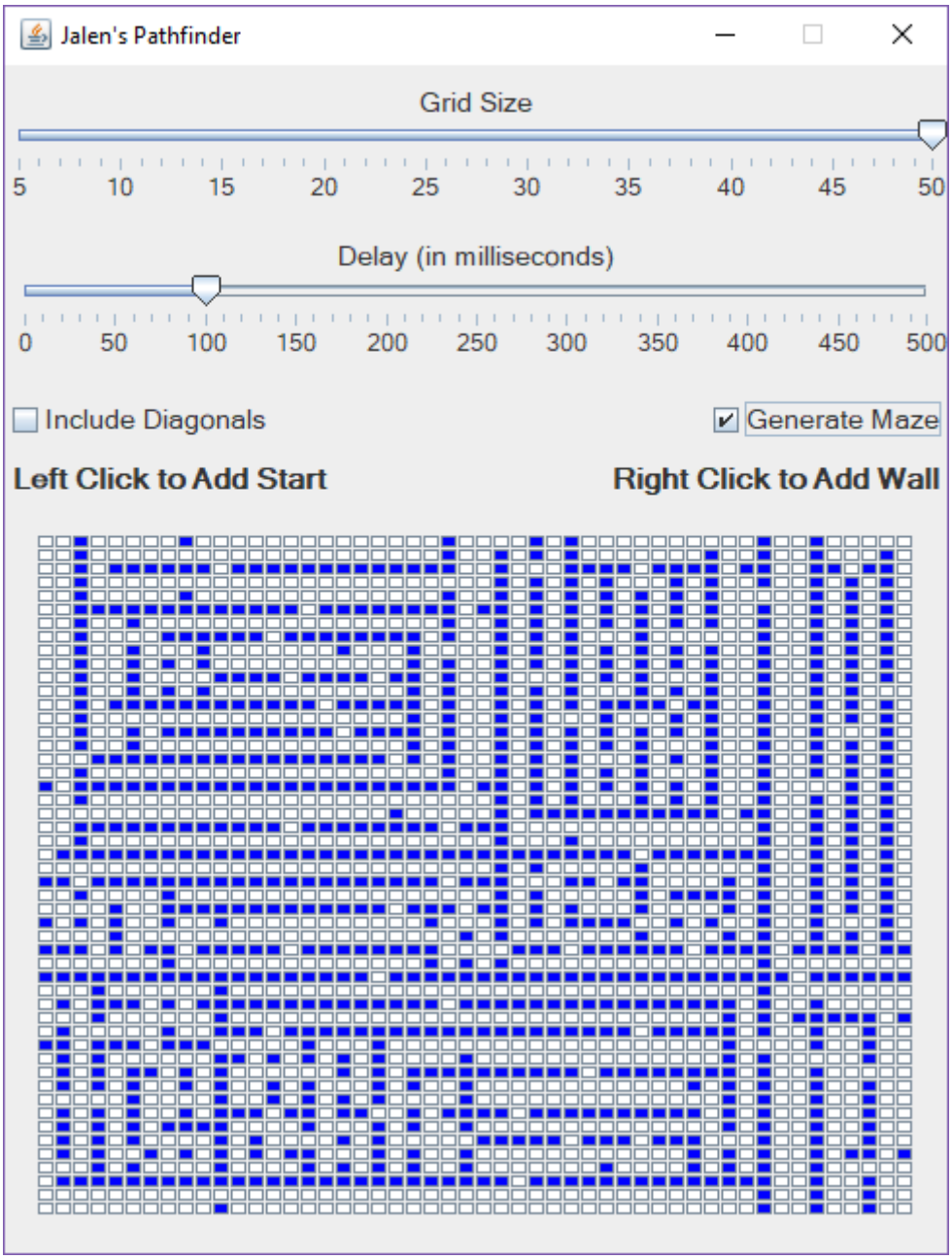
Found then Drawn Shortest Path



Start Node Moved to End Node's Spot



Dragged “Grid Size” Slider to 5 cells



Dragged “Grid Size” Slider to 50 cells

Checked “Generate Maze” Box