```java
package maze.solvers;

import java.text.DecimalFormat;

/**
 * A-Star (Lee) algorithm for maze solving
 *
 * @author Pierre-Andre Mudry, Romain Cherix
 * @date February 2012
 * @version 1.2
 *
 */
public class AStar {

    private MazeElem[][] maze;
    private int width, height;
    private int[][] solution;

    // Debug information
    public final boolean VERBOSE = true;

    private AStar(MazeContainer mazeContainer) {
        maze = mazeContainer.maze;
        width = mazeContainer.nCellsX;
        height = mazeContainer.nCellsY;
    }

    /**
     * Solves the maze
     *
     * @param x The x-coordinate of the start point
     * @param y The y-coordinate of the start point
     */
    private void solve(int x, int y) {
        /**
         * The solution at the beginning is an array full of zeroes
         */
        solution = new int[width][height];

        // We indicate the starting position
        solution[x][y] = 1;

        // This is the step counter
        int m = 1;

        /**
         * Do the expansion until we have reached the exit.
         */
        while (expansion(m) == false) {
            m++;
        }

        /**
         * m contains the total number of steps to find the solution
         */
        if (VERBOSE)
            System.out.println("\n[AStar solver] Took " + m + " steps for the solution\n");

        /**
         * As the forward propagation is over, we can now do the back-prop
         * phase.
         */
        backtrace(m);
    }

    /**
     * Lee forward propagation algorithm
     *
     * @param m The current step of the algorithm
     * @return A boolean value that indicates if wave has hit exit
     */
    private boolean expansion(int m) {

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

                /**
                 * At each step m, we propagate the wave for each cell of the
                 * solution that has the index m.
                 */
                if (solution[i][j] == m) {
                    if (!maze[i][j].wallWest)
                        if (maze[i][j].isExit) {
                            return true;
                        } else if (solution[i - 1][j] == 0)
                            solution[i - 1][j] = m + 1;

                    if (!maze[i][j].wallNorth)
                        if (maze[i][j].isExit) {
                            return true;
                        } else if (solution[i][j - 1] == 0)
                            solution[i][j - 1] = m + 1;

                    if (!maze[i][j].wallEast)
                        if (maze[i][j].isExit) {
                            return true;
                        } else if (solution[i + 1][j] == 0)
                            solution[i + 1][j] = m + 1;

                    if (!maze[i][j].wallSouth)
                        if (maze[i][j].isExit) {
                            return true;
                        } else if (solution[i][j + 1] == 0)
                            solution[i][j + 1] = m + 1;
                }
            }
        }

        return false;
    }

    /**
     * Grants uniform access for the whole maze and makes sure that we do not
     * cross the borders of the maze
     *
     * @param i x position
     * @param j y position
     * @return distance to the origin point, -1 if outside the graph
     */
    private int access_solution(int i, int j) {
        if (i >= width || i < 0 || j >= height || j < 0)
            return -1;
        else
            return solution[i][j];
    }

    /**
     * Lee algorithm back-trace phase when the array has been annotated with the
     * distances
     *
     * @param m The highest distance from origin point
     */
    private void backtrace(int m) {
        int[][] ret = new int[width][height];

        int x = 0, y = 0;

        // Get the coordinates of exit in original maze
        for (int j = 0; j < height; j++) {
            for (int i = 0; i < width; i++) {
                if (maze[i][j].isExit) {
                    x = i;
                    y = j;
                    break;
                }
            }
        }

        // The exit is part of the solution
        ret[x][y] = 1;

        /**
         * While we haven't reached the beginning, annotate the solution with
         * the correct path
         */
        while (m > 0) {
            if (access_solution(x - 1, y) == m && !maze[x][y].wallWest)
                ret[--x][y] = 1;

            if (access_solution(x, y - 1) == m && !maze[x][y].wallNorth)
                ret[x][--y] = 1;

            if (access_solution(x + 1, y) == m && !maze[x][y].wallEast)
                ret[++x][y] = 1;

            if (access_solution(x, y + 1) == m && !maze[x][y].wallSouth)
                ret[x][++y] = 1;

            m--;
        }

        // Update the solution with the backprop version
        solution = ret;
    }

    /**
     * Displays the solution on the console for control
     */
    public static void displaySolution(int[][] mazeSolution) {
        String solutionText = "";

        int width = mazeSolution[0].length;
        int height = mazeSolution.length;

        if (mazeSolution != null)
            for (int j = 0; j < width; j++) {
                for (int i = 0; i < height; i++) {

                    DecimalFormat myFormatter = new DecimalFormat("00");
                    String s = myFormatter.format(mazeSolution[i][j]);

                    if (i != height - 1)
                        solutionText += s + " - ";
                    else
                        solutionText += s;
                }
                solutionText += "\n";
            }

        System.out.println(solutionText);
    }

    /**
     * This class is thought to be used statically using only this method
     *
     * @param mc The {@link MazeContainer} that we want to solve
     * @param x The x-coordinate of the start point
     * @param y The y-coordinate of the start point
     * @return An array containing 1's along the solution path
     */
    public static int[][] solve(MazeContainer mc, int x, int y) {
        AStar alg = new AStar(mc);
        alg.solve(x, y);
        return alg.solution;
    }

    public static void main(String args[]) {
        /**
         * Create a maze and display its textual representation
         */
        MazeContainer mc = new MazeContainer(50, 80);
        TextDisplay.displayMaze(mc);

        /**
         * Compute a solution and display it
         */
        int[][] solution = AStar.solve(mc, 0, 0);
        AStar.displaySolution(solution);

        GraphicDisplay gd = new GraphicDisplay(mc, 2, false);
        gd.setSolution(solution);

    }

}
```