```java
package maze.solvers;

import maze.data.MazeContainer;

/**
 * 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.
         */

        // TODO When you are confident your algorithm is working, you
        // can uncomment this line in order to have the backtrace done for you
        // 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) {

        // TODO Implement your algorithm here

        return true;
    }

    /**
     * Grants uniform access for the whole maze and makes sure that we do not
```

```
89      * cross the borders of the maze
90      *
91      * @param i
92      *            x position
93      * @param j
94      *            y position
95      * @return distance to the origin point, -1 if outside the graph
96      */
97     private int access_solution(int i, int j) {
98         if (i >= width || i < 0 || j >= height || j < 0)
99             return -1;
100        else
101            return solution[i][j];
102    }
103
104    /**
105     * Lee algorithm back-trace phase when the array has been annotated with the
106     * distances
107     *
108     * @param m
109     *            The highest distance from origin point
110     */
111    private void backtrace(int m) {
112        int[][] ret = new int[width][height];
113
114        int x = 0, y = 0;
115
116        // Get the coordinates of exit in original maze
117        for (int i = 0; i < width; i++) {
118            for (int j = 0; j < height; j++) {
119                if (maze[i][j].isExit) {
120                    x = i;
121                    y = j;
122                    break;
123                }
124            }
125        }
126
127        // The exit is part of the solution
128        ret[x][y] = 1;
129
130        /**
131         * While we haven't reached the beginning, annotate the solution with
132         * the correct path
133         */
134        while (m > 0) {
135            if (access_solution(x - 1, y) == m && !maze[x][y].wallWest)
136                ret[--x][y] = 1;
137
138            if (access_solution(x, y - 1) == m && !maze[x][y].wallNorth)
139                ret[x][--y] = 1;
140
141            if (access_solution(x + 1, y) == m && !maze[x][y].wallEast)
142                ret[++x][y] = 1;
143
144            if (access_solution(x, y + 1) == m && !maze[x][y].wallSouth)
145                ret[x][++y] = 1;
146
147            m--;
148        }
149
150        // Update the solution with the backprop version
151        solution = ret;
152    }
153
154    /**
155     * Displays the solution on the console for control
156     */
157    public static void displaySolution(int[][] mazeSolution) {
158        String solutionText = "";
159
160        int width = mazeSolution[0].length;
161        int height = mazeSolution.length;
162
163        if (mazeSolution != null)
164            for (int j = 0; j < width; j++) {
165
166                for (int i = 0; i < height; i++) {
167
168
169                    if (i != height - 1)
170                        solutionText += mazeSolution[i][j] + " - ";
171                    else
172                        solutionText += mazeSolution[i][j];
173                }
174                solutionText += "\n";
```

```
175             }
176
177         System.out.println solutionText ;
178     }
179
180     /**
181      * This class is thought to be used statically using only this method
182      *
183      * @param mc
184      *            The {@link MazeContainer} that we want to solve
185      * @param x
186      *            The x-coordinate of the start point
187      * @param y
188      *            The y-coordinate of the start point
189      * @return An array containing 1's along the solution path
190      */
191     public static int[][] solve MazeContainer mc, int x, int y  {
192         AStar alg = new AStar mc ;
193         alg.solve x, y ;
194         return alg.solution;
195     }
196
197     public static void main String args[]  {
198         /**
199          * Create a maze and display its textual representation
200          */
201         MazeContainer mc = new MazeContainer 4, 4 ;
202         TextDisplay.displayMaze mc ;
203
204         /**
205          * Compute a solution and display it
206          */
207         int[][] solution = AStar.solve mc, 0, 0 ;
208         AStar.displaySolution solution ;
209     }
210
211 }
```