# Grid Space Simulator

## Problem statement:

- Build a simulator that initializes a start state, a goal state, and some obstacles
- Implement a path planning policy to reach the goal state from the start state
- Improve the simulator, so it breaks the path planner
- Boost the path planner to have it conquer the new improved simulator

## Steps along the way:

1. Initial thoughts (v0.1):
    a. Use OpenGL libraries written by my instructor at a course at CMU to help visualize
    b. Use a grid based solution, where each point is either free space, or occupied by an obstacle – the free space points can be traversed, the obstacles cannot
    c. Use a graph to represent the grid, each point represented by a node
    d. Once I have the graph, I can implement the Dijkstra algorithm to find the quickest path
    e. Therefore, implement a Node class, a Graph class, and a Solver class, which interact with each other and implement the above ideas

2. Inputs, and improvements suggested (v1):
    a. Refrain from third party libraries, so a decision was made to use a console representation of the grid
    b. Replace the Node and Graph classes by simpler Point and Grid classes, solely for the purposes of visualizing on console
    c. Implement the Graph within the Solver class, whose default constructor takes a Grid object and uses its information to generate the Graph

3. First raw implementation, and lessons learned towards v2:
    a. Use a 2D Point array for the grid representation, use public getters and setters to build the grid
    b. Too time consuming and pointer intensive (sigh!) to implement a graph in the Solver class – drop the Dijkstra approach for something better
    c. A* search seems better! It doesn't need to explore the whole map, and makes decidedly goal oriented progress decisions; plus since it focusses on the goal, it doesn't need to explore the whole grid, like the Dijkstra algorithm => runtime improvement expected

4. Towards an efficient solution (v3):
    a. Use a 1D array to represent the 2D array of Points – very useful, since we now need to deal only with a single index
    b. Separate the simulator from the solver – ergo: the birth of the Grid class and the Solver class

c. Have the Solver class take in a Grid object, and work its magic – this separates the grid setup from the path finding process; we can now have the grid built as per need, and when satisfied, pass it to the heavily protected (almost everything private) Solver class

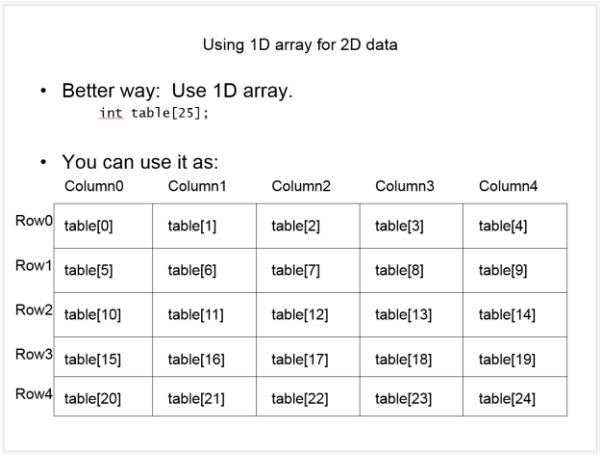
5. Auxilliary decisions:
   a. Representing the grid: The grid is visible to the user in a fashion shown below (whitespace represents free traversable area):

```
####################
#            #     #
#      #     #     #
#      #     # <   #
#   >  #           #
#      #           #
####################
```

# - Outer wall or obstacle
> - Start state
< - Goal state
* - Step in the discovered path (not visible in the schematic above)

   b. Indexing the grid: The grid is zero-indexed, and numbered starting from the top left corner; when the row finishes, the next index is assigned to the first point of the next row. I'm very used to this idea, representing a 2D array using a single 1D int array in memory, and I picked it up from my professor. The grid is numbered much like:

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |




My instructor's method!

6. Implementing the ideas discussed above:
    a. Classes:

```
class Point
{
private:
        int x, y;    // coordinates of the point
        int idx;     // index to reference into the 1D point array
        bool obstacle;      // is this point an obstacle?
        bool solution_path;         // does this point lie on the solution path?
};
```

```
class Grid
{
private:
        int x_len, y_len;      // dimensions of grid
        int num_gridpoints;         // number of grid points on this grid
        int num_obstacles;// number of points which are obstacles (including walls)
        Point* points;      // a dynamically allocated array of points which make this grid up
        int start_index, goal_index;  // to keep track of start and goal states
};
```

```
class Solver
{
private:
        int x_len, y_len;      // dimensions of grid
        int num_gridpoints;         // number of grid points on this grid
        int num_obstacles;// number of points which are obstacles (including walls)
        int start_index, goal_index;  // to keep track of start and goal states
        Point* points;      // a dynamically allocated array of points which make this grid up

        Solver(Grid grid)    // constructor

        int* open_f_values;
        int* open_g_values;
        int* parent_tracker;
        int* closed_set;
};
```

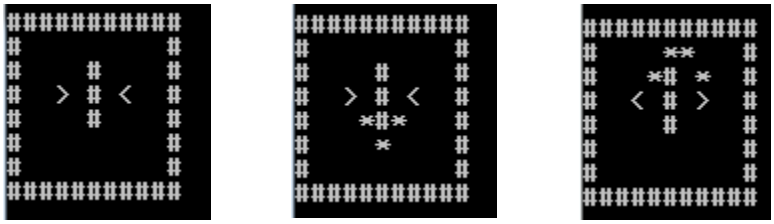
    b. Usage:
        i. The user creates a grid object, and then has various methods available, to populate the grid
        ii. The constructors handle all edge cases, so there are no memory leaks even if illegal or careless inputs are used
        iii. By default, the grid is always created with an immutable bounding wall to prevent 'falling off'
        iv. Once created, the user can then add obstacles, and also define the start and goal state locations (if not provided as input, start and goal states are always defined by default); the grid can then be displayed using the consoleDraw() method
        v. Finally, once the user is satisfied with the grid, a solver object is initialized: this is highly encapsulated – it has a one statement solve routine, and also has a consoleSolutionDraw() method, which outputs the path policy to get from the start state to the goal state, using the A* search algorithm to get there

c. Complexities:
    i. For the most part, constructors are called once the grid size has been specified, this means that there is, *at best* $O(n)$ (n = number of grid points) time complexity, since the constructors need to populate the dynamic arrays with some initial values
    ii. However, all code except the A* search itself, is also, *at worst*, of $O(n)$ complexity, which has been accomplished by representing much complex data structures as sparse arrays, so the good news is that we have a $\Theta(n)$ time complexity (except for the solver itself)
    iii. The same, unfortunately, cannot be said for the space complexity, which is also of the order of $O(n)$ – the objects themselves, have member variables which are this size, and the Solver object makes wasteful use of space by initializing and maintaining as many as 4 different dynamic arrays; this can probably be ironed out given more time, as I can then come up with even leaner data representations to store all this information
    iv. Next, to offset the space usage, I've refrained from using the conventional tools to make the A* search work – for example, the heap which maintains the F scores for the nodes to be visited next in ascending order, has been replaced with a much simpler array representation, the `parent_tracker` array; further, the need for maintaining a heap has itself been done away with, because at each iteration of the algorithm, this implementation checks to see the lowest F value, and simply stores it separately, forgetting the others – ensuring that the next most eligible point is picked up automatically, without querying a heap
    v. And finally, to address the elephant in the room, the complexity of the search algorithm itself – theory dictates that this value is bounded by $O(b^d)$, and in this case, the branching factor $b$ is 8 (worst case 8 neighbors to search at each state), with $d$ being the distance to the goal state; in my application of the same though, this is a much worse bound than what is practically seen, because I use good heuristics to prune away many of those branches which are guaranteed to not lead to the goal

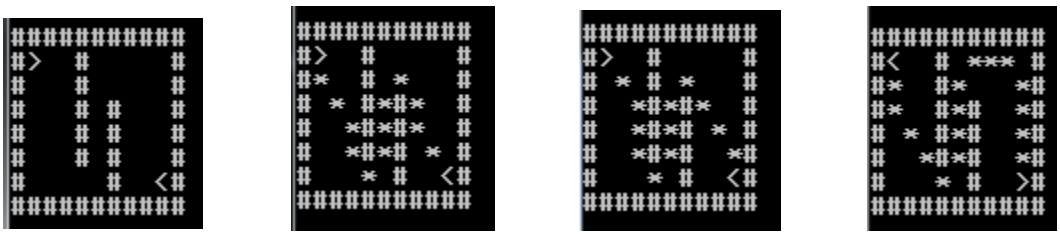
7. Flow of the code:
    a. Phase 1: SIMULATOR: A nominally sized grid is first selected, with one or two simple obstacles; further, the start and goal states are well defined, either by default, or by the user
    b. Phase 2: SEARCH ALGORITHM: The simple A* algorithm crawls through the grid, and spits out the path policy which needs to be taken
    c. Phase 3: BEEFING UP THE SIMULATOR:
        i. I found that the search generally performed sub-optimally when there were obstacles which forced the robot to initially move farther from the goal state (sometimes even failing to find the solution); further, obstacles placed too closely, which had the solution path passing between them, were generally not discover, and also resulted in failure
        ii. I therefore implemented a new kind of obstacle course, one which had exactly one solution: through a criss-crossed path
        iii. This new weapon in the arsenal of the simulator absolutely pulverized the search algorithm; its speed also went way down
    d. Phase 4: REVITALISING THE SEARCH ALGORITHM:
        i. This called for really going back to the drawing board for changing many nuts and bolts in the search algorithm

ii. My first issue was the sub-optimal path generated when forced to move farther from the goal state: this was fixed by penalizing any changes in direction (smoothing), and soon, it started spitting out shorter paths to the goal state
iii. To fix the issue of not detecting narrow paths through obstacles, I used varying penalty values to see if I could coax to solver to get there – not to be underplayed, this process was *extremely* frustrating!
iv. Until finally, an idea struck me: running the algorithm multiple times in a loop, with ever increasing penalty values – this caused the algorithm to better itself on its own, as it searched for the least cost path, until it finally settled on the sweet spot value
v. Which brings me to the time consumed: running the algorithm with multiple values of penalty pushed its performance down even farther! Luckily, Wikipedia came to my rescue, and I implemented the concept of bounded relaxation (weighted A*) – using a bloated value of the heuristic function to cause the algorithm to take faster, albeit less efficient jumps
vi. And with those tools in my belt, I set about finding the weight and the penalty values which would give the best results for a particular grid setup
vii. And those tools, I give to you to tweak: my main function has a detailed comment which explains how to vary these two parameters (penalty and heuristic_weight) to get to the goal state presently: not too slow, and not too sub – optimally

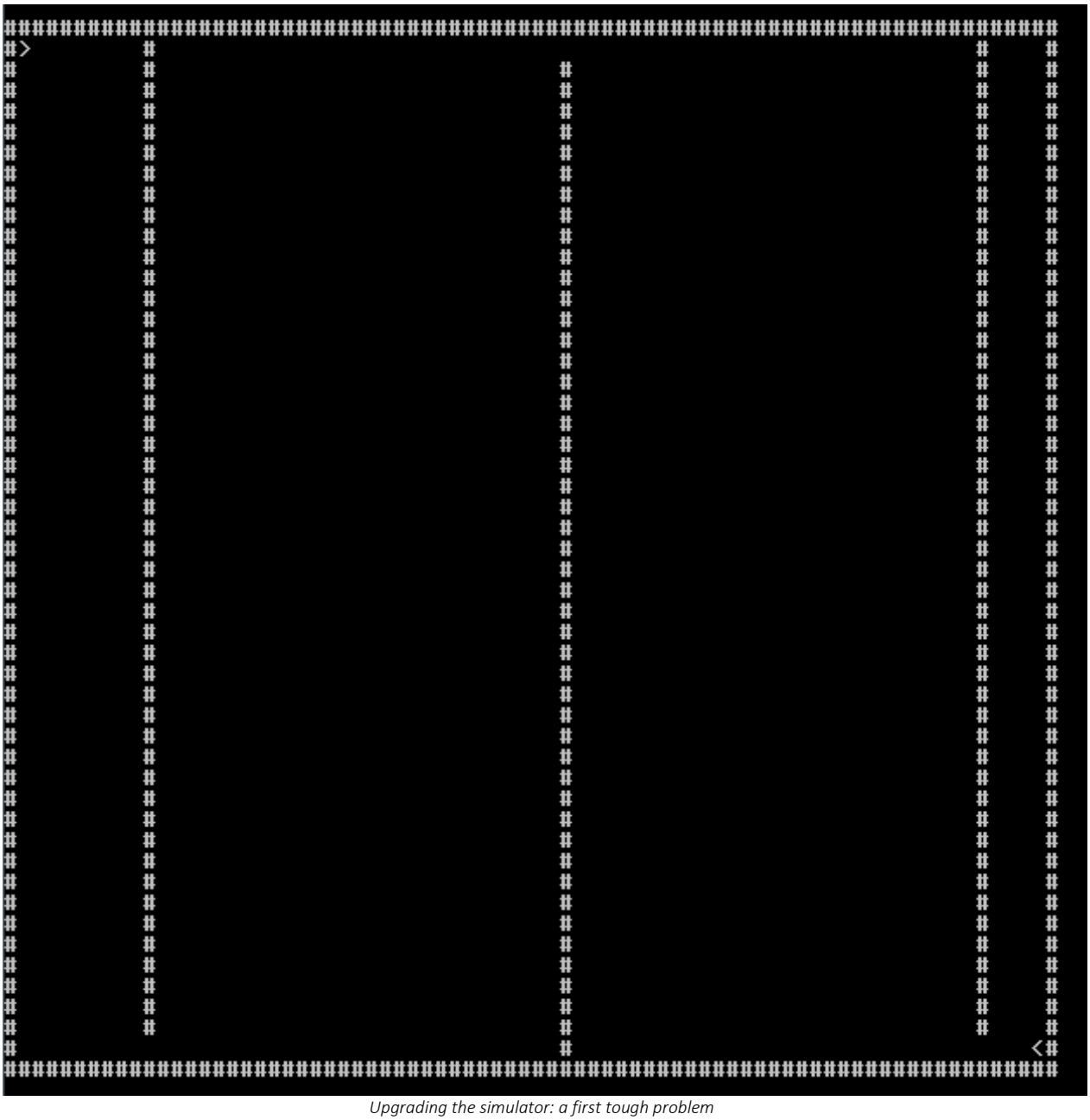
8. Gallery (here's a sample of my results):



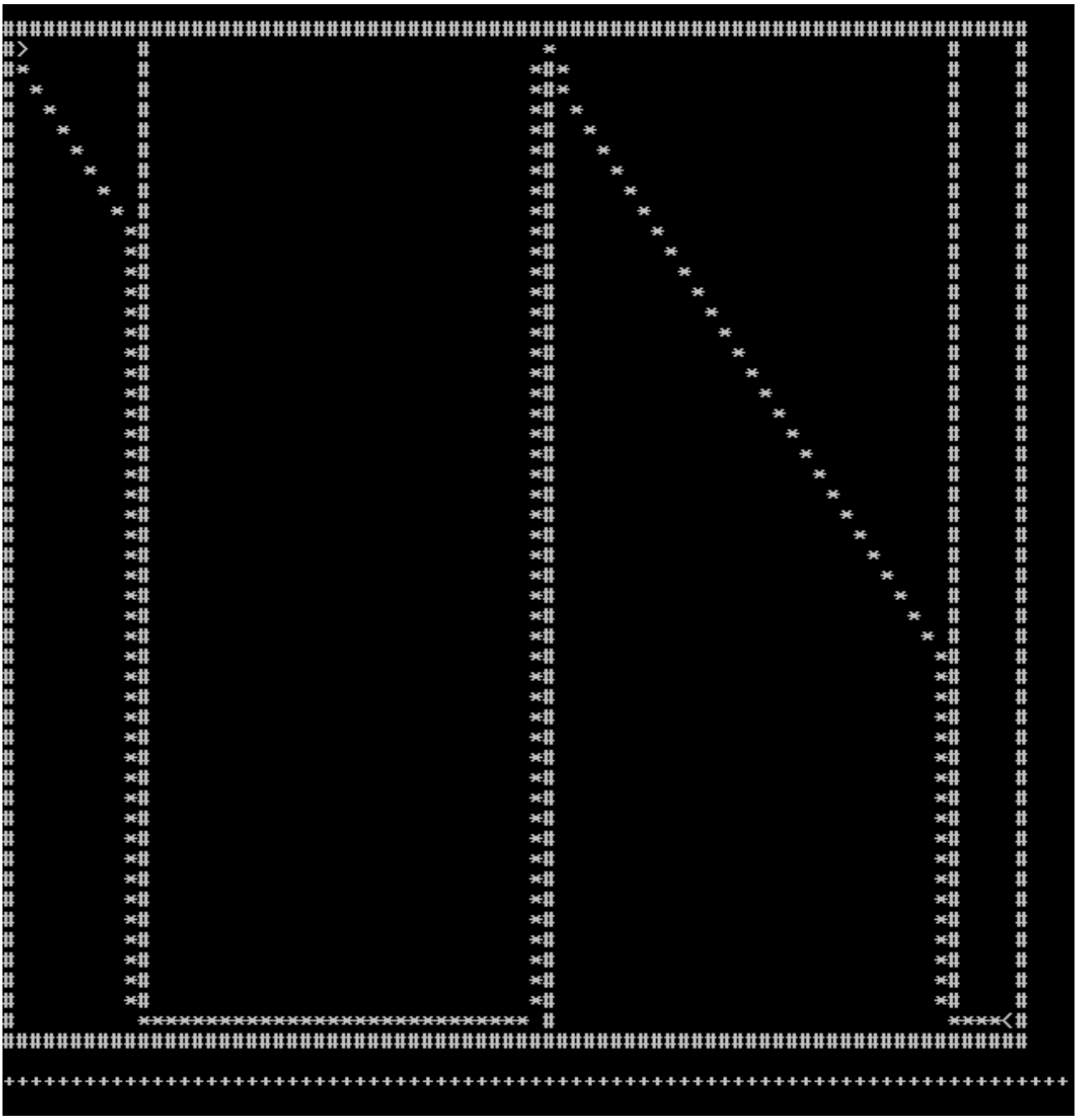
*Building the simulator: The first simple problem and its solution*
*a) Problem, b) Simple solution, c) Solution upon switching the states*



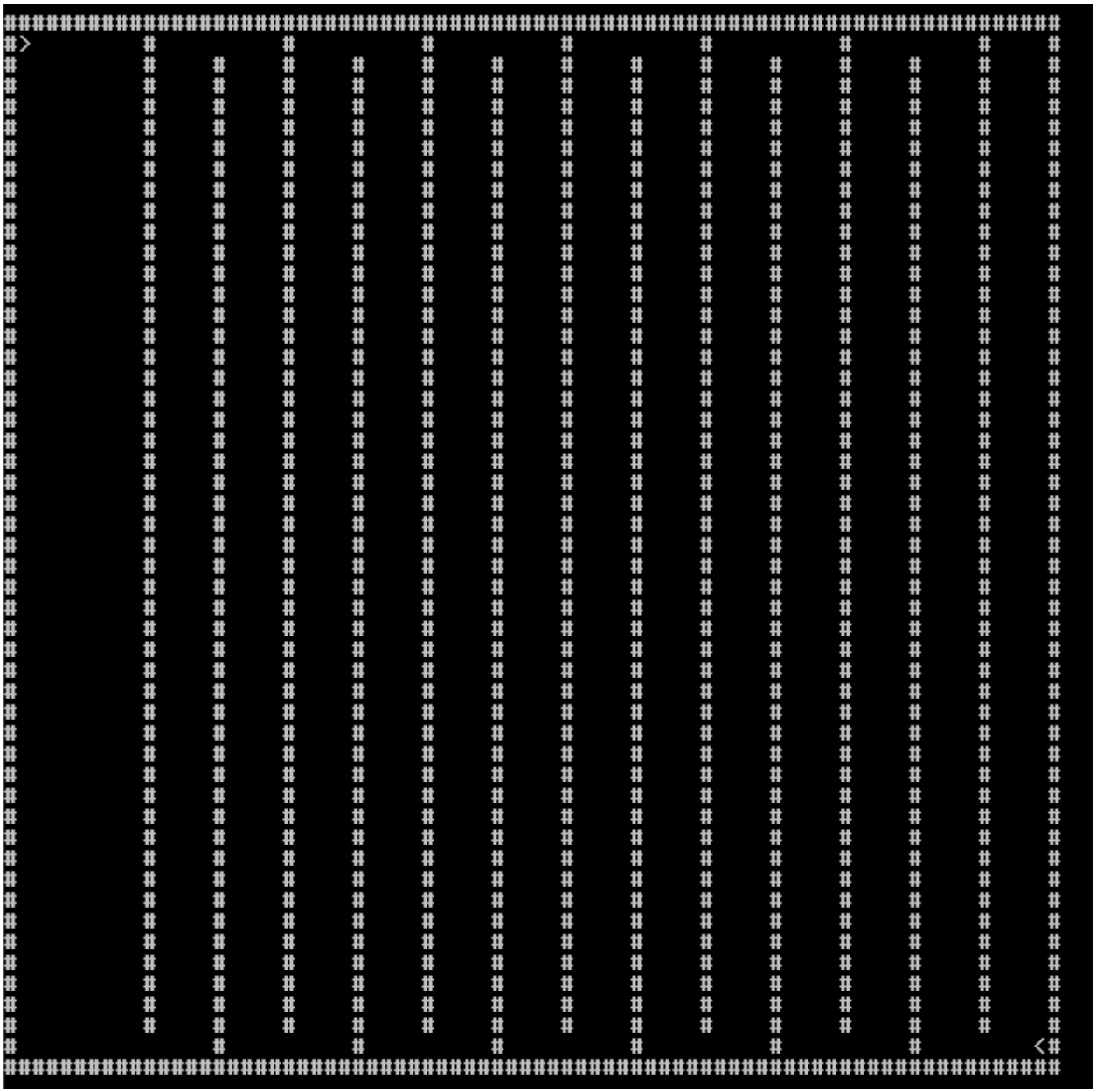
*Another problem, which broke the solver, because of the narrow path*
*a) Problem with no simple solution, b) Solution upon penalizing b) Solution upon stricter penalizing, d) Solution upon switching the states*



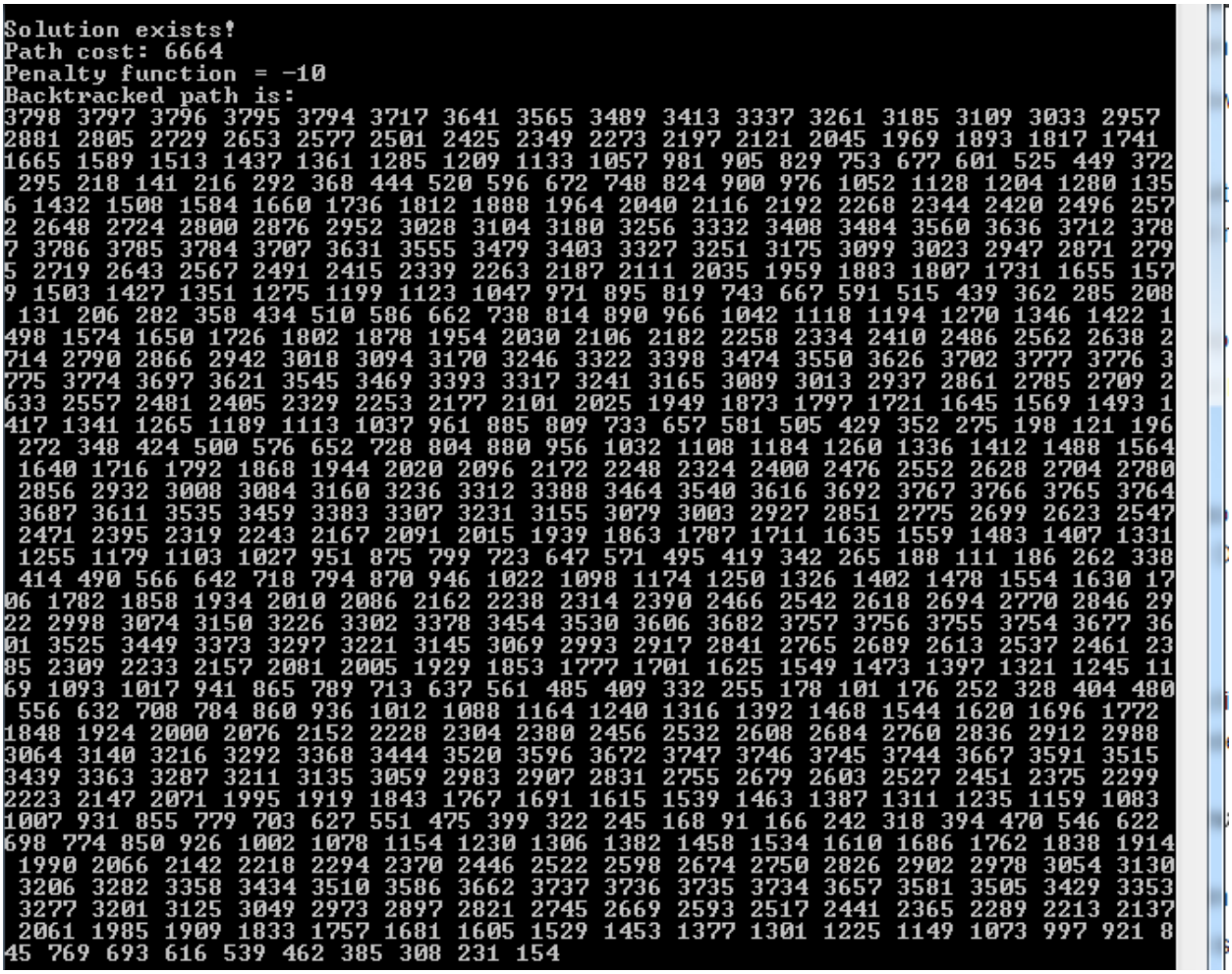
*Upgrading the simulator: a first tough problem*
*Notice the relatively lower freedom for the solver to search through*



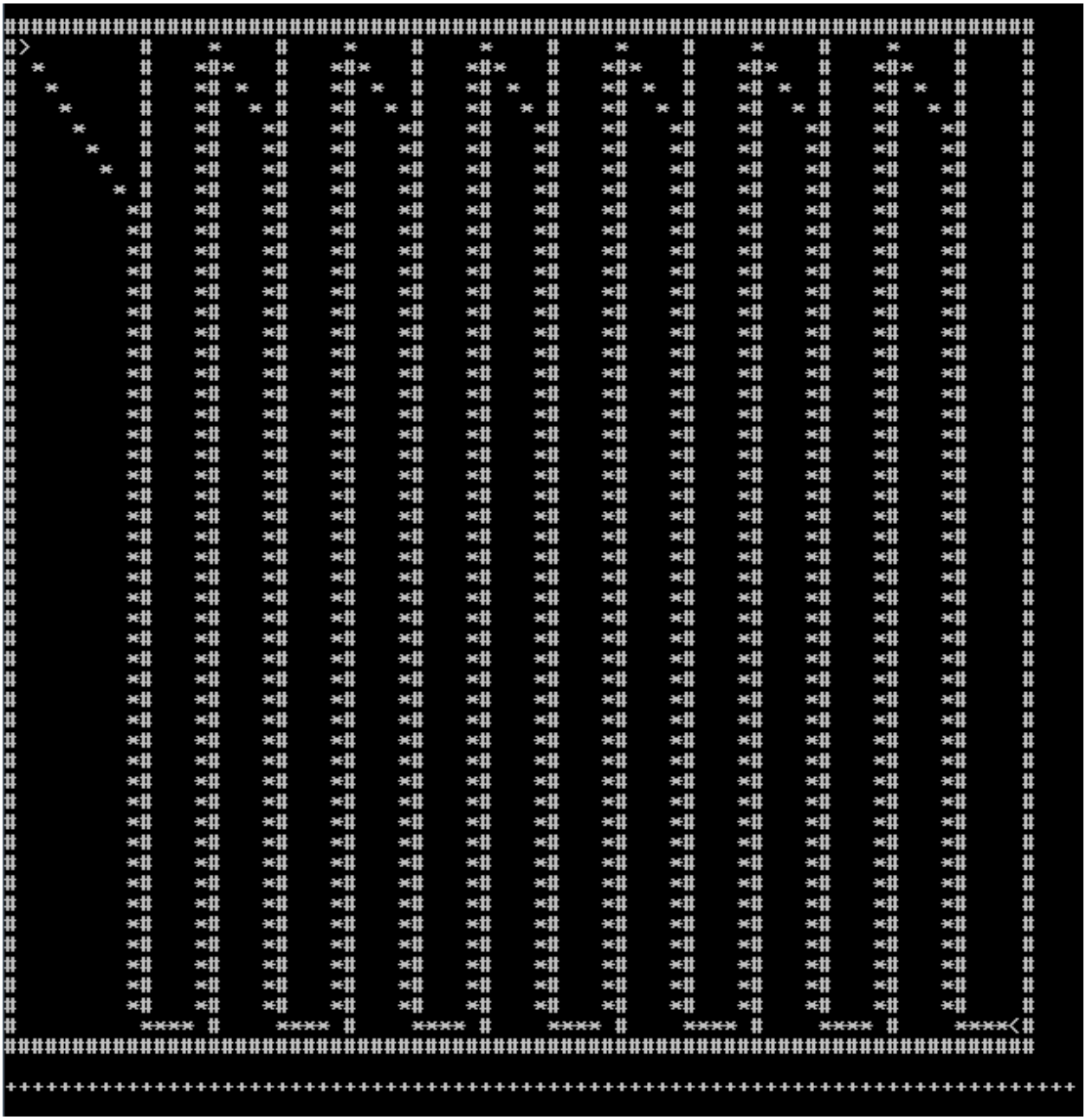
*Upgraded the solver, but it had to be penalized, or it took weird sub-optimal routes*



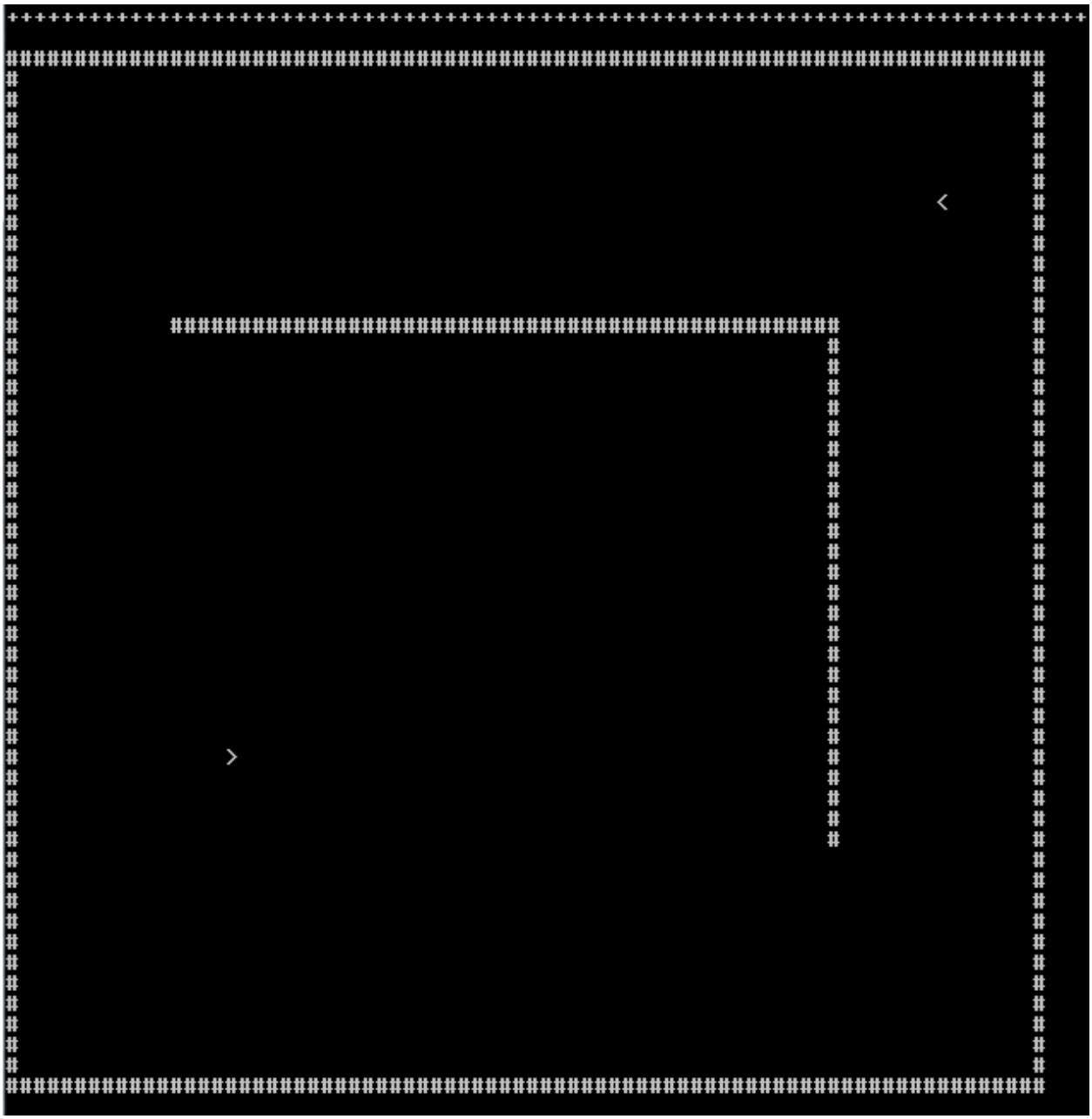
*A much more demanding problem, dubbed the beast problem, with relatively little wiggle room*



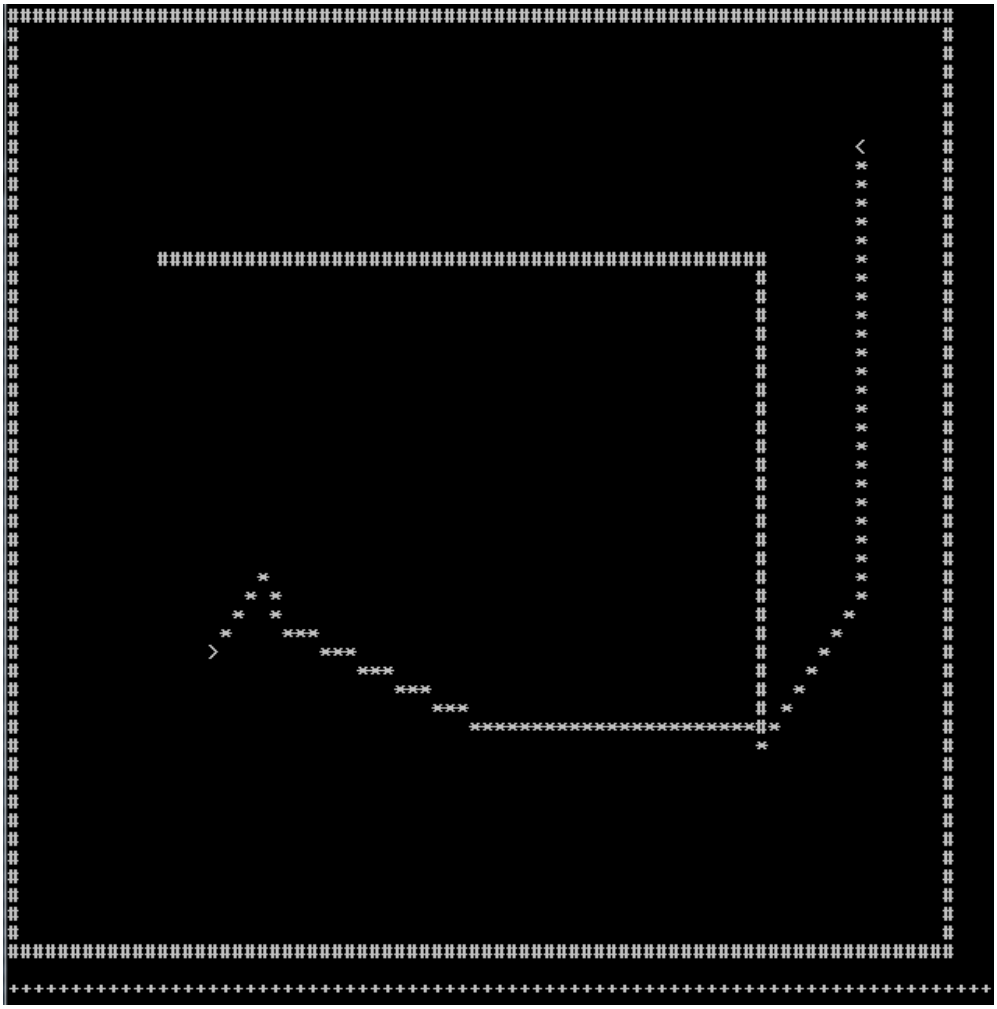

*And voila! It was solved, albeit a bit too slowly*



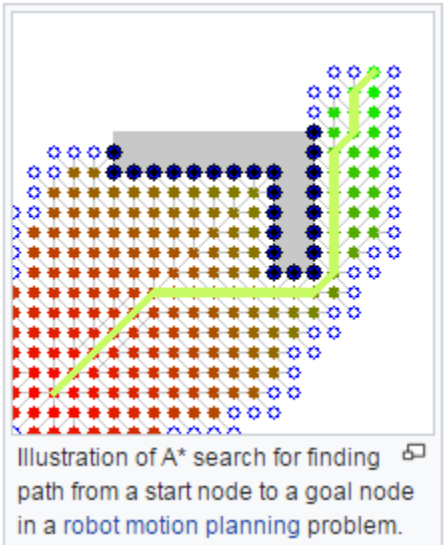
*Solution of the beast problem, although it took a lot of time*
*The good news was that even if I tightened the wiggle room even further, the algorithm iteratively found how best to penalize itself to still make it through!*



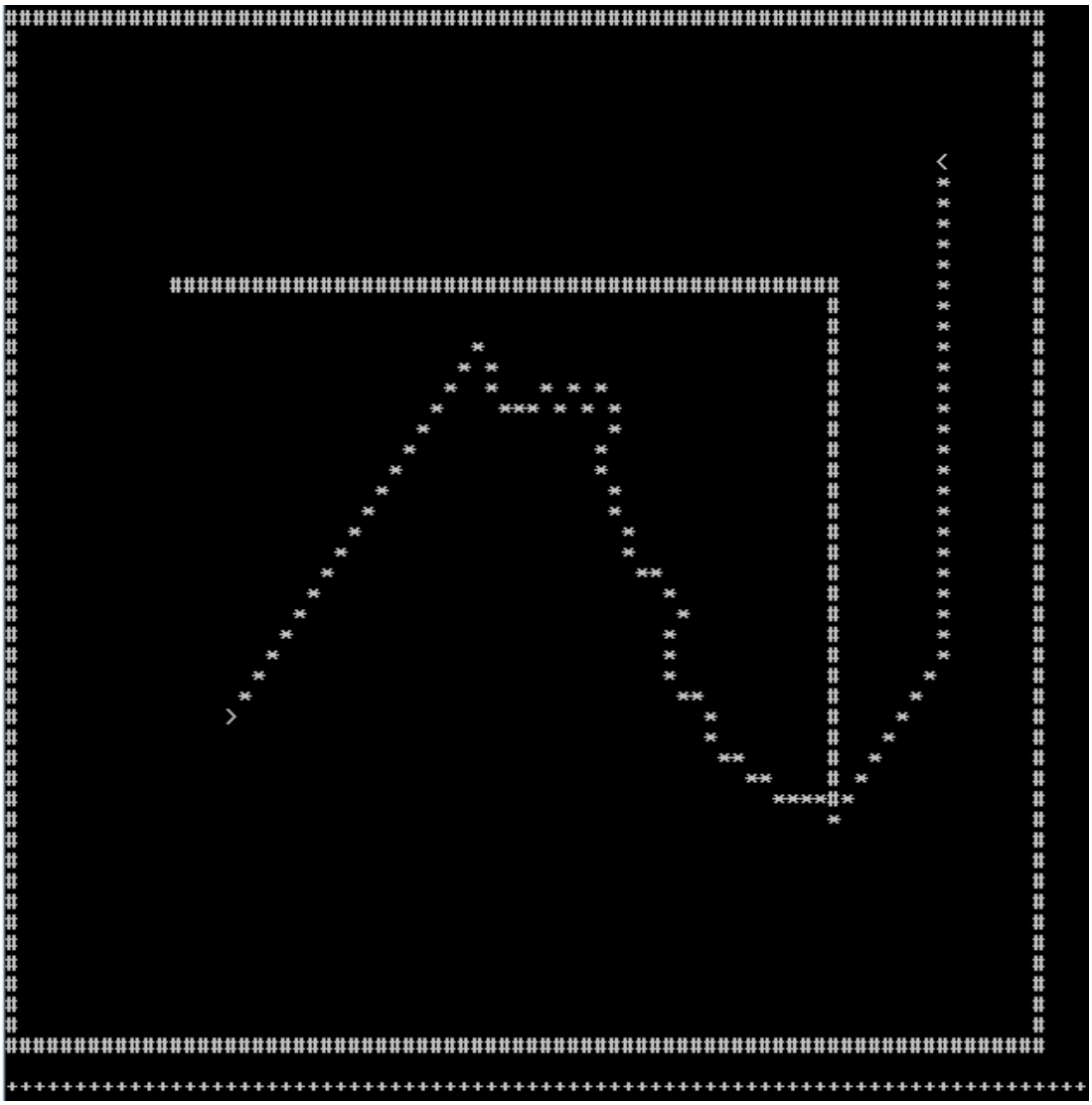
*Trying to mimic wikipedia's sample problem on the A* page (https://en.wikipedia.org/wiki/A*_search_algorithm)*



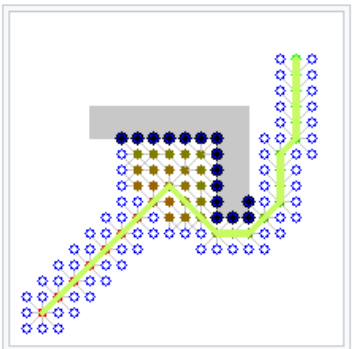
*Simple search solution (which is slow), with self penalizing approach*



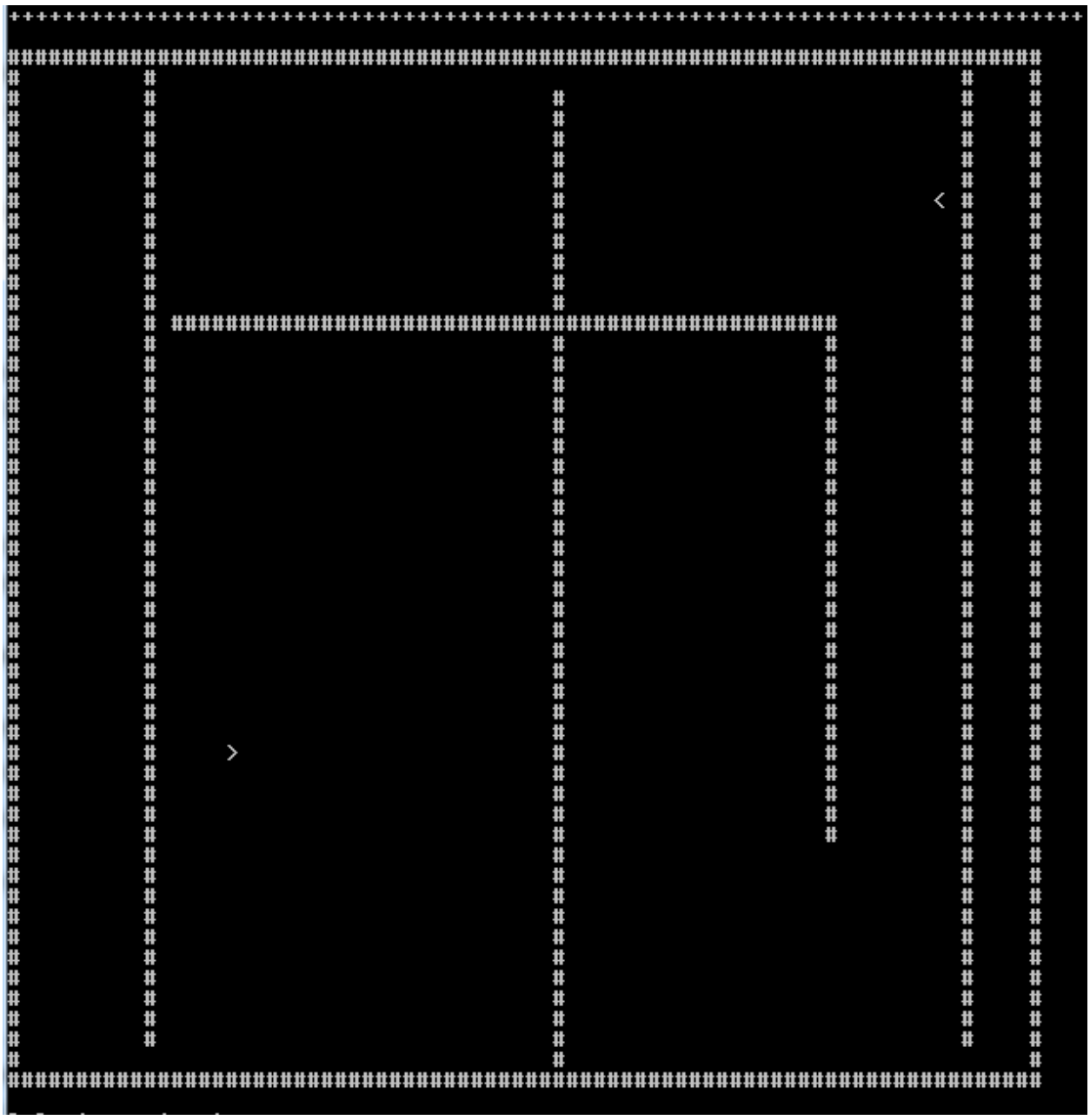

*Wikipedia's interactive illustration of the same*
*Matches mine, doesn't it?*



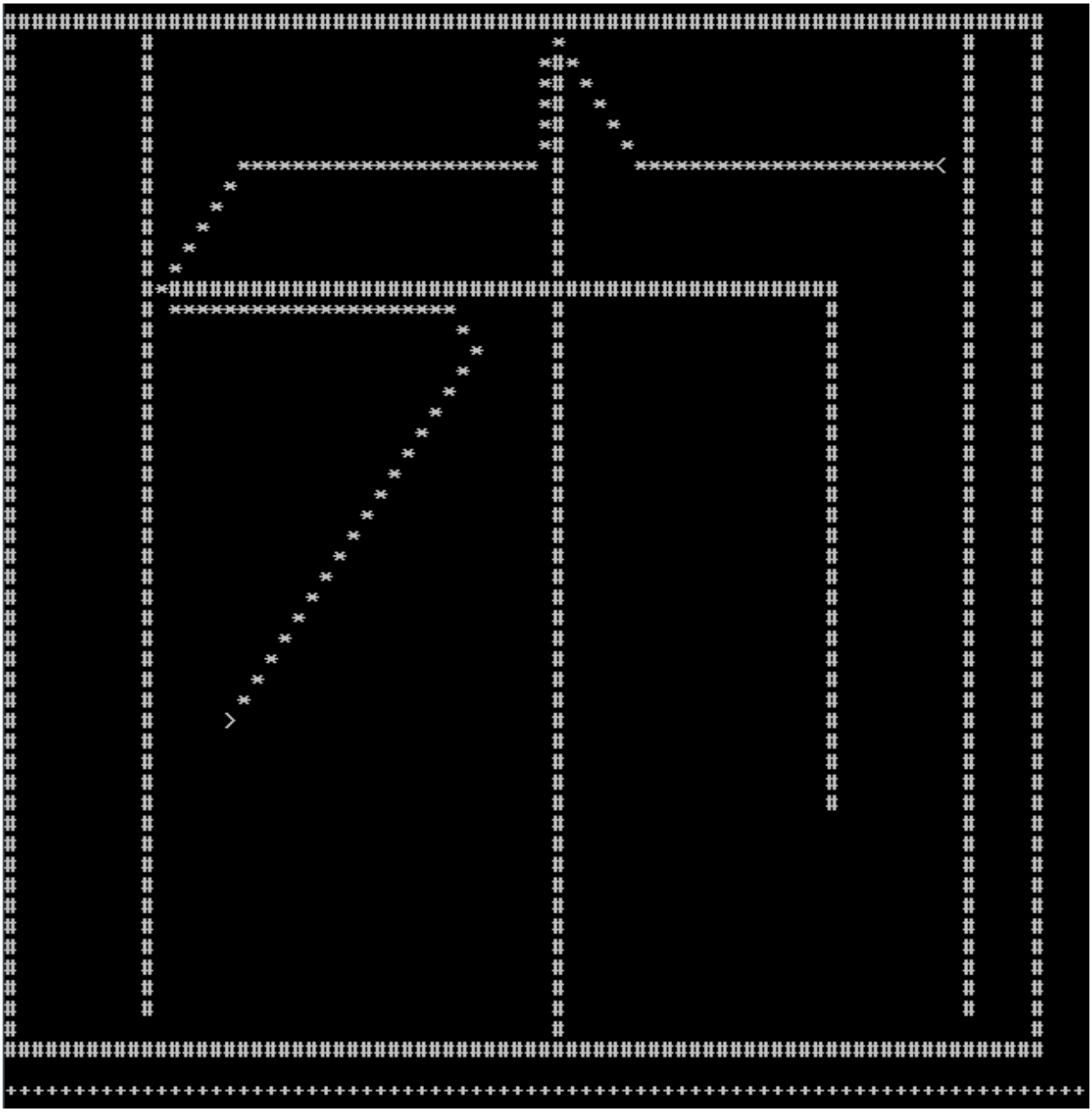
*Speeding up the heuristic by weighting it (in addition to penalizing the algorithm), in exchange for costlier paths*



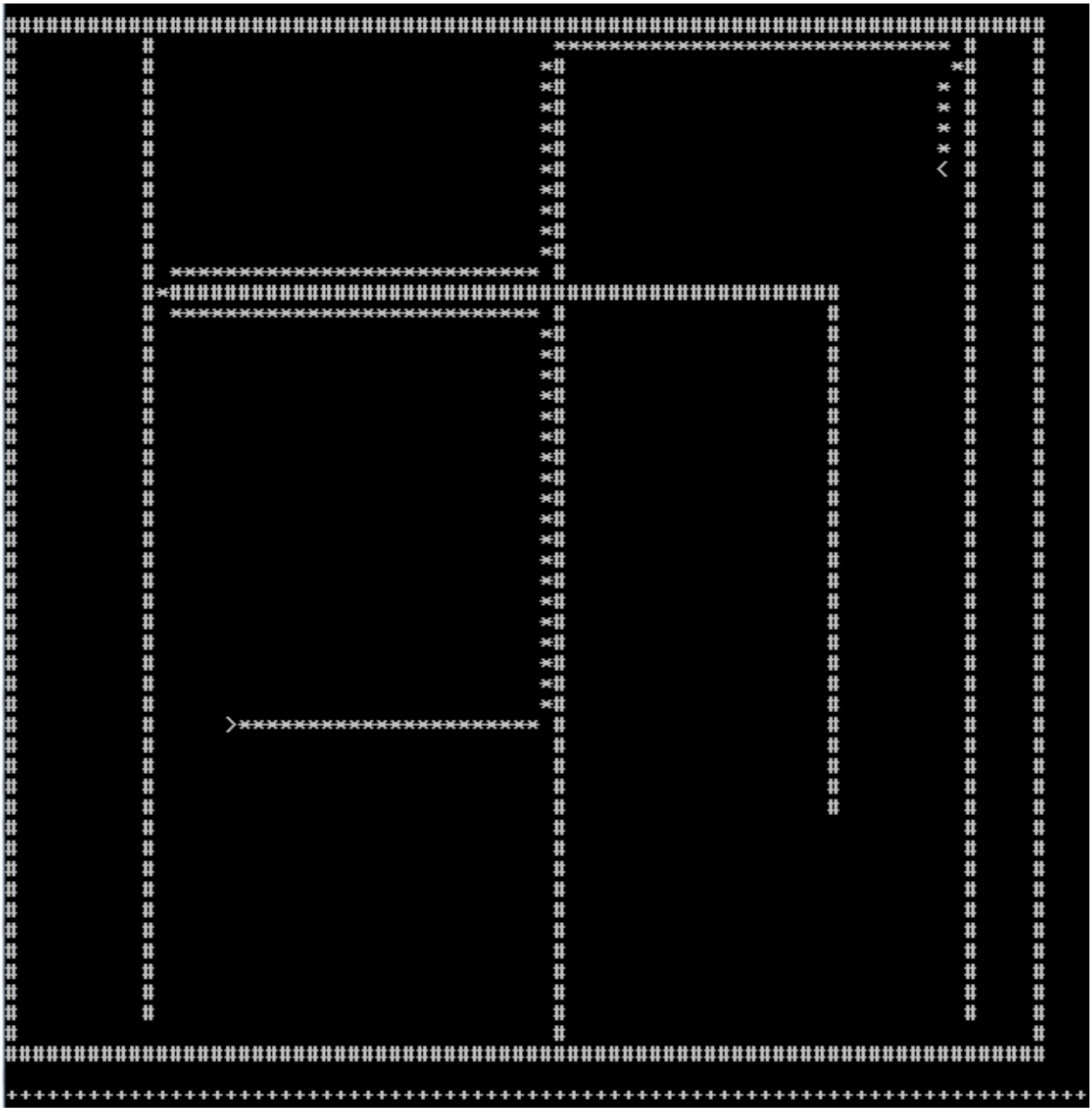

*Wikipedia's interactive illustration of the same*
*This one matches as well!*



*A more complex problem*

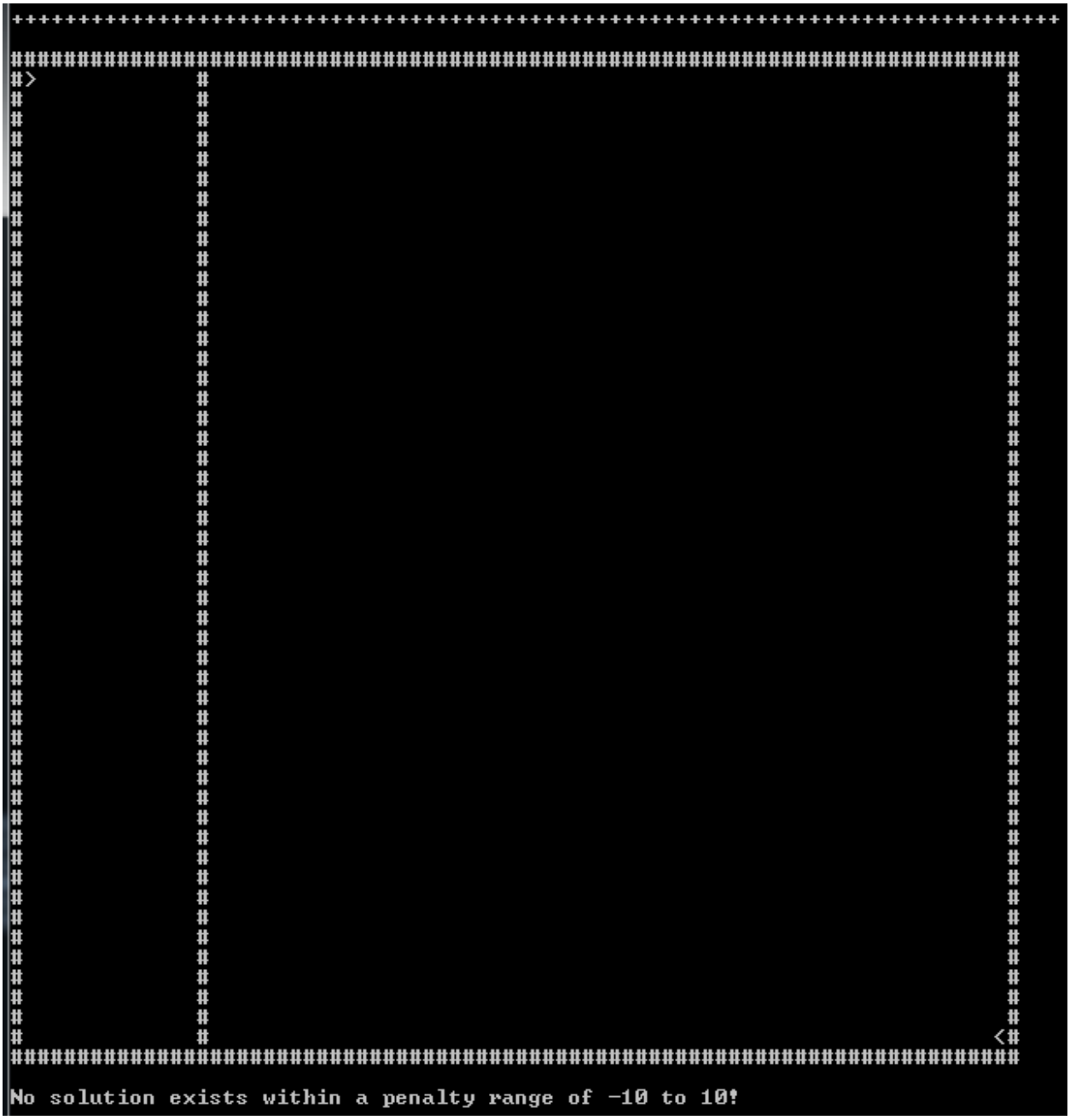


*Solution with unweighted heuristic, but penalized to walk in a straight line when possible*



*Solution with weighted heuristic (double the Manhattan distance): notice how constrained the solution became*
*The solver failed to solve this problem with a triple weighted heuristic, because at that point, maybe it was being too greedy, and looking too far ahead, to realize that it had already failed to spot the tiny passage on the left*

*Looks like a swastika (I honestly had no hand in that ☺)!*




*And finally, here's a sanity check: Of course it fails when there is no path!*

9. Final thoughts:
    a. The code challenge was a huge learning experience!
    b. As of my tests, it solved 98% of all grids I threw at it, and the ones it failed at were the ones where it was too ambitious, and used a highly weighted heuristic
    c. Suffice to say that this should work on most grids with obstacles, indeed, not only that, it will intelligently adjust itself to find either cheaper, or quicker paths, as instructed
    d. There's some cleanup needed in terms of extra data structures which I put in, in anticipation of future needs, so that should free up some memory
    e. And, as is always true with algorithms in general, I conclude this challenge by encouraging you, to ponder over the age old question – Can I do better? Can I do faster?

Thanks for taking the time to go through my implementation! Hope you learned something!