Project: A*

In this project you're going to write a route finder. The applet you write should be able to read in map files, and using a variety of different heuristics to shape the search, display the shortest path between the start and finish squares, if any. (If there are multiple equally-short paths, any one of them can be displayed.)

Part 1: GUI, grid, animation

The navigable space

The "board" will be a rectangular grid of arbitrary size; at each cell of the grid, you can have one of four possible things: the start, the finish, an impassable wall, or a plain old open space.[1]

The format for files containing a grid requires the first line to consist of two integers, representing the width and height of the grid respectively. Each subsequent line represents one row of the grid, with each cell represented by a single char: a lowercase 'O' for the start space, an asterisk for the finish, a hash mark ('`#`') for a wall, or a period for an open space. For example:

```
6 5
......
...*..
..##..
......
..o...
```

You can assume that every correctly-input grid will have exactly one start and exactly one finish. Extra lines in the file are ignored (and thus are a convenient place to write comments about that particular grid).

Is there a path?

In the first phase of this project, we simply need to determine whether there exists a path from start to finish. A simple breadth-first search, managed with a queue, will do fine for now; later, this part will get more sophisticated, of course.

[1] There will actually be one more cell type, the teleporter, which will be discussed anon.

The display thereof

To display the actual grid (and eventually paths on the grid), there needs to be a GUI. Walls are displayed as solid black squares, open squares should be white, and the start and target squares are clearly marked. During the solving process, squares that have been explored turn grey. and if a route is found, the squares that lie along that path should have small (but visible) black dots drawn in the middle of them.

The control widgets let you specify a grid file and load it in; a way to start the solver; and a way to either take a single step or to start animating the search process.

Task: Grok the code

In the course directory, you will find a solution for the above problem description. This will be your starting point for future work, so copy it into your own directory.

Read through this code and make sure you understand it fairly thoroughly.

Part 2: Pathfinding

As written, the task and program above were written only to determine whether a path existed; they do not do any of the work of actually recovering that path, and in particular, of recovering the *shortest* such path.

Modify the code so that once a path is found, all the grid locations along that path are marked in the middle with a black dot. Don't forget that since this is an event-based GUI, you won't be able to simply draw a dot once, but rather, you'll have to store the path information in such a way that every time `paint()` gets called, the path will be drawn.

Hint: the data structure you use to keep track of path information while seeking a path in the first place, and the data structure you use to actually store and draw the path once it's found, need not be the same.

Part 3: Faster pathfinding

To shape the search for a shortest path from start to finish, implement the A* algorithm. It should be efficient. For your first pass, you can assume a default heuristic of $h(X) = 0$ (this is equivalent to the breadth-first search from above), but see below for other heuristics you'll need to be able to run with. Remember that your implementation will just do one step of the algorithm at

a time, rather than implement as a `while` loop. (The provided code already meets these requirements; just make sure not to break them as you edit!)

When the algorithm finishes, the interface should clearly indicate whether there is, or is not, a path, and if there is, you should reconstruct it and draw it as described above.

NB: as you implement the following subproblems, you don't have to retain the various interfaces used in the existing code base. If you need to add new methods or change headers, feel free.

Heuristics

Confirm that your code works with a simple queue-based agenda before trying to implement the heuristics; once you're ready, though, you should implement agendas whose policies are organised according to each of the following four heuristics:

- $h(X) = 0$
- $h(X) = E(X) =$ Euclidean distance between X and the finish, truncated to next lower integer
- $h(X) = M(X) =$ Manhattan distance between X and the finish
- $h(X) = P(X) =$ "Proximity sensor metric" to finish:
  - $P(X) = M(X) \quad \text{if} M(X) < 8$
  - $P(X) = 8 \qquad \text{if} M(X) \geq 8$

The GUI should let a user start the search with any of the four metrics, either using four separate Start buttons or by adding a pull-down list.

Heuristic tiebreakers

Once you've implemented all the heuristics and have tried them on a variety of test cases, you may have noticed that even with an informed heuristic, the algorithm still is exploring a lot of space it "obviously" shouldn't have to. Resolving this flaw is not trivial, but neither is it especially difficult: it involves retooling your agenda policy so that if the main utility function $f(X) = g(X) + h(X)$ computes equal utility for two different locations, you use a modified utility function to break the tie. Implement this; add a checkbox to the GUI, and if it is checked when the solver starts, make your agenda policy break ties when the utility function turns up equal values.

Teleportation

Add another square type to the input, represented by the character `'@'`. If present in the input at all, there will be exactly two of them; they represent teleporters. A teleporter behaves exactly like an open square, with one exception: from one teleporter, the other teleporter is exactly one step away (just as if they were adjacent). A run given the input

```
13 3
......#......
.@.o..#..*.@.
......#......
```

should successfully find a path from start to finish via the teleporters. Furthermore, since each teleporter is just one step away from the other, the input

```
13 3
.............
.@.o.....*.@.
.............
```

should still find a path via the teleporters—that route has four steps exclusive of the endpoints, while the non-teleporter shortest route has five.

Note that teleportation is not obligatory; a run on the input

```
13 3
.............
..o.@.*...@..
.............
```

should find a shortest path that cuts right across the first teleporter without involving the second one at all.

In addition to your code...

Don't forget that you need a README file that tells me how to run your code.

In your electronic handin, include an appropriate number of test maze files. They should both test the algorithm for correctness and highlight the strengths and weaknesses of the different heuristics. Include comments in the files as to which of these goals are being accomplished.

The squares that are shaded grey—i.e. those that were discovered in the course of the search—collectively show the "shape" of a search, and this shape is affected by which heuristic was used. As a separate paper handin or in a plain text

file in your electronic handin, select a test maze where all four heuristics yield different search shapes. Explain why the shapes look the way they do.

On paper or in a plain text file, discuss how the existence of teleporters affect admissibility of the various heuristics. For any that remain admissible, sketch a proof as to why. For those that do not, give an example input that "breaks" it.

On paper or in a plain text file, discuss how the heuristic tiebreaker affects admissibility of the various heuristics.