Project 3: A*

Due: 2 November 2017

In this project you're going to write a route finder. The program you write should be able to read in map files, and using the A* algorithm with different heuristics to shape the search, display the shortest path between the start and finish squares.

Objectives

In the course of this project, the successful student will:

- implement a standard best-first pathfinding algorithm,

- write a custom hash function and use a hashtable to keep track of explored items,

- use a map/dictionary to maintain information to reconstruct the path upon completion, and

- write a custom comparator and use a priority queue to manage the agenda of items to explore.

Expected input and output

The program you write will look for one or two **command line arguments**. The first is required, and specifies the file from which to read the board. The second is optional, but if present, specifies what kind of search heuristic to use (once you get to the part where that's relevant).

The input files will be in this general format:

```
8 5
........
..#*....
..#.....
..#.####
..#.o...
```

Specifically, each file will contain a rectangular grid of arbitrary size, preceded by a line with two positive integers representing the width and height of the grid (respectively). At each cell of the grid, you can have one of four possible things: the start ('o'), the finish ('*'), an impassable wall ('#'), or a plain old open space ('.').

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

The output of the program should end with a display of the grid and an information/status line. This final output grid should be just like the input grid except:

- If a path exists from start to finish, the shortest such path is marked with asterisks, and

- whether or not a path exists, all of the locations that have been explored (and aren't marked with asterisks) are marked with hyphens.

If there are multiple equally-short paths, any of them is fine; the start node may be marked with either an o or an * if a path is found, whichever's more convenient for you.

The information line at the end should say either:

- "`No path.`", or

- "`Path found with length` n.", replacing n with the length of the shortest path.

During the running of the program, you can display the grid and current status as many times as you like; for these intermediate states the status line can be something other than the two lines given above.

Prep work

For this project, I'm not giving you any code to start from. Your initial prep work involves getting the outer shell of the program running: getting the command line argument(s), reading the grid, leaving a stub to process the grid and look for paths, and printing the grid and final status.

In the initial prep work, the "look for paths" part will be minimal: it's only required to find paths with length 1, i.e. those where the start and finish are literally adjacent to each other. In cases where they're further apart, it's acceptable (in the prep checkpoint) to report no path. In cases where there is actually no path, the program should report no path.

Note that your program should not crash or give strange output if the start or finish are directly adjacent to the edge of the grid, as in the example above!

This work is due a week from Thursday (19 Oct) **at 4pm**. You'll have a chance to ask questions about it in class on Thursday to clear up any last-minute issues, but you really don't want to wait until then to start it. When you're ready to hand it in, use the `handin` script with the assignment name `proj3`.

Design work

Once you've got the shell of the program running (or perhaps while you are working on that, but the design work depends on course content we won't cover until next week), you can start thinking about the algorithm and data structure design. I want to focus in particular on four aspects to plan for:

1. Write pseudocode for a function `shouldExploreBefore(a,b)` that takes two coordinate pairs and answers the question, "should we explore location a before location b?". This will make use of the equation $f(X) = g(X) + h(X)$ that we'll be discussing in class next week, but your pseudocode will need to be more specific than that. How do you keep track of g? What about h?

2. Work out the detailed numbers on paper for a particular example. Specifically, draw out a sample grid that isn't one of the ones we did in class, where the shortest path requires at least one detour around a wall (that is, the shortest path is longer than the Manhattan distance from start to finish); and label every square that will ever be in the agenda with that square's f, g, and h values.

3. Write out some test cases. Draw grids that aren't ones we've done in class, which collectively enumerate meaningfully different test cases. Remember that test cases require us to say both what the input to a

system is, and what we expect the result to be. You should certainly have more than two test cases, but you'll have to think carefully about how many are needed here to effectively test the different things you need to test. For each test case you write out in your notes, remember to annotate it with what's special about that case (i.e. why you've included it).

4. Think about the data structures. In the objectives for this project, I mentioned three standard data structures—how will they be used? For each of the three, indicate where in the algorithm you'll add values to the data structure and where you'll access or remove from the data structure.

Write your design work on paper (or do it on your laptop, but paper's probably easier for this) and bring it to class; this work is due **at the start of class** on Tuesday the 24th. If you're really stuck on something, do your best, make a note of it, and move on; we'll be discussing this extensively in class.

Final version

A full-credit final version will be a complete, non-buggy, working implementation of the A* algorithm TOGETHER WITH convincing proof that it is correct. The program should be able to find the correct path on any input, and be able to apply admissible heuristics based on both Euclidean and Manhattan distances (chosen from the command line at runtime); and when the main heuristic results in a tie in deciding which locations to explore next, the algorithm should break the tie in an appropriate way. The "proof" will take the form of a clean and complete set of test cases, including both input and expected results

Note that I am not able to spend a ton of time with your program (and in fact may not read it at all, and definitely won't do your debugging for you), so your documentation will need to tell me anything I need to know to run and test your program. There need to be clear instructions on how to run it in general as well as how to run each/all of the tests and quickly verify that they ran correctly (and which rubric items each one corresponds to). Having complete and correct documentation is an easy 25 points, but if your documentation omits important info or tells me the wrong thing, you'll get less than full credit there.

After prep work (15 points), design work (10 points), and documentation (25 points), there remain 100 points in the rubric, which will be awarded according to the table below. Under each score, I show (for your convenience) the total cumulative points if you get that item plus *all* the previous points, and the letter grade this corresponds to. It is arranged roughly in the order I suggest you attempt them, with the earlier ones being easier or enlightening with respect to the later ones, but you can in general get points for the later ones (if they work) without getting the earlier ones.

NOTE: if your code doesn't compile, or immediately crashes when it's run, you will get zero of these points. Don't let this happen to you!

Score	Description
10 (60/D−)	Runs on all valid inputs without crashing, and output matches requirements for prep work (at least). Don't forget to test non-square grids.
5 (65/D−)	Can store, query, and iterate a set of "explored" locations (either printing them out as coordinate pairs or by successfully completing the "shows explored" rubric item).
5 (70/D)	Successfully explores more than just the adjacent locations without ever falling off edges (crashing or bad access) or walking through walls.
10 (80/D+)	Shows "explored" locations in displayed final grid. (NOTE: You can do this before anything listed further down—*and you should*. Getting a visual on what your algorithm is doing is immensely helpful.)
5 (85/C−)	Explores substantially outward from start, repeatedly finding and exploring additional adjacent nodes as managed by an agenda.
10 (95/C)	Reports whether path exists, for all valid inputs.
10 (105/B−)	Reports length of path that was found (whether or not this was the shortest path).
10 (115/B)	Reconstructs a path that was found (whether or not this was the shortest path), showing it either as sequence of coordinate pairs or by successfully completing the next rubric item.
5 (120/B+)	Shows the path that was found in the displayed final grid (whether or not shortest).
5 (125/B+)	Found path is always a valid shortest path, for all inputs.

Score	Description
10 (135/A)	Exploration is shaped by a shortest-first heuristic based on the (correctly-computed) Manhattan distance.
10 (145/A+)	User can choose on the command line between Manhattan and Euclidean distance measures, and Euclidean heuristic is correct.
5 (150/A+)	When Manhattan-based heuristic yields a tie between two locations, the algorithm uses an effective tiebreaker to search more efficiently.

Don't forget that there is an implicit "and prove it" after all the rubric items. This is especially salient for the always-shortest-path line, and for the shaped-by-Manhattan line—choose some test cases that illustrate that you have done these things, because if the only way to know it's correct is by reading the code, you won't get the points.

Tips

You're welcome to write this project in any language that runs on torvalds and the lab systems. (And if you want a language that's not there, but could be installed, let me know and I can make that happen.)

The prep work might seem familiar to you. Feel free to refer to work you did in other classes to get that running. If it's not familiar to you and you don't know how to do something, ask! That stuff is not what this project is really about, and I don't want you losing a lot of time on it.

In the early phases of implementation, use a stack or queue to hold your agenda, before you get the priority queue stuff working.

You almost certainly don't want to modify the internally-stored grid object after you've read it in. Instead, use other data structures to store information, and if you print out the grid, use that extra information to choose what to print.

If you're not sure what Manhattan distance and Euclidean distance are, look them up or ask. The Euclidean distance formula is probably the one you'd think of if you know a "distance formula", but Manhattan distance you might not have seen before.

Handing in

For both the prep work and the final version, hand it in as `proj3` using the handin script. It is due at 4pm on the due date.